AFIT/GSS/LAS/93D-3

ADA275961

THE DEVELOPMENT AND USE OF AN EVALUATION MECHANISM FOR THE ASSESSMENT OF SOFTWARE CONFIGURATION MANAGEMENT TOOLS

THESIS

Wayne M. Descheneau, Captain, USAF
Neil W. Robinson, Captain, USAF

AFIT/GSS/LAS/93D-3

012
200

94-05457

Approved for public release; distribution unlimited

94 2 18 060

Best
Available
Copy

Disclaimer

The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

Accession For

NTIS GRA&I ✓
DTIC TAB ☐
Unannounced ☐
Justification

By
Distribution/

Availability Codes

Dist	Avail and/or Special
A-1	

Preface

The purpose of this research effort was to help Air Force managers in the Air Force 1) comprehend the SCM requirements contained in Department of Defense standards and 2) understand how commercially available automated SCM tools can help meet specific SCM requirements on a software development effort. To accomplish this goal we developed an evaluation mechanism, the heart of which was a matrix built with SCM requirements on the vertical axis and common tool functionality on the horizontal axis. This cross reference presented in the matrix will identify which tool functions are used (if any) by a given tool to meet each SCM requirement of interest. This evaluation mechanism was used to assess two commercially available SCM tools as a test to determine its effectiveness.

We extend our gratitude to those who assisted us during this effort. First, we thank our thesis advisors, Mr. Dan Ferens and Capt Brian Holmgren, for their guidance and encouragement throughout this research effort. We appreciate the support provided by Julie Kingsbury from TRW in cooperation with the Software Technology Support Center at Hill AFB, Utah. Her efforts saved hours of investigation and headaches by providing tool vendor points of contacts and getting us moving in the right direction. We also thank Mrs. Joan "Zip" Robinson for her copy editing skills and copious review of our final draft thesis.

Most importantly, I, Wayne, appreciate the support of my family. I express my deepest gratitude to my wife, Torie, for her understanding while I was locked away in my study and the confidence that she inspired within me. I thank Bacchus, my Alaskan Malamute, for his playfulness that kept me sane by telling me when it was time to take a break.

Wayne M. Deschenau

Neil W. Robinson

Table of Contents

List of Figures

List of Tables

Acronyms

ACSN	Advance Change Study Notice
CALS	Computer aided Acquisition and Logistics Support
CCB	Configuration Control Board
CDR	Critical Design Review
CI	Configuration Item
CM	Configuration Management
CPIN	Computer Program Identification Number
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DID	Data Item Description
DoD	Department of Defense
ECP	Engineering Change Proposal
EMD	Engineering and Manufacturing Development
ERR	Engineering Release Record
FCA	Functional Configuration Audit
HWCI	Hardware Configuration Item
IDD	Interface Design Document
IRS	Interface Requirements Specification
PCA	Physical Configuration Audit
PDR	Preliminary Design Review
SCI	Software Configuration Item
SCM	Software Configuration Management

SCN	Specification Change Notice
SDD	Software Design Document
SDF	Software Development File
SDL	Software Development Library
SDP	Software Development Plan
SDR	System Design Review
SPS	Software Product Specification
SQPP	Software Quality Program Plan
SRR	System Requirements Review
SSR	Software Specification Review
STD	Software Test Description
STP	Software Test Plan
STSC	Software Technology Support Center
TRR	Test Readiness Review
UDF	Unit Development Folder
VDD	Version Description Document
WBS	Work Breakdown Structure

AFIT/GSS/LAS/93D-3

Abstract

This research effort investigated the development of a mechanism for use in the evaluation of automated Software Configuration Management (SCM) tools. An examination of applicable DoD standards identified the SCM requirements that could be levied on a development contractor at the time of this writing. A literature search revealed the functionality common to the various automated tools that were commercially available. These two sets of information were organized and compiled to form a matrix, having rows comprised of the SCM requirements and columns comprised of the tool functionality . For each requirement that was met, the intersection on the matrix of the requirement and each functionality used to meet that requirement, in full or in part, was checked or marked. This matrix was identified as the Requirements-Functionality matrix and comprised half of the evaluation mechanism. The other half consisted of general information about a given tool and an area to substantiate each requirement identified as being met by the particular tool. The evaluation mechanism was then used on two commercially available SCM tools: Aide-De-Camp and the Product Configuration Management System. The purpose was to 1) refine the evaluation mechanism, 2) to determine the effectiveness of the evaluation mechanism, and 3) evaluate the chosen tools. The results revealed which areas of SCM were focused upon and which functions were used most often for each tool. The results also revealed that a thorough understanding of each tool's capabilities was required in order to complete the evaluation mechanism. The evaluation mechanism prescribes a method for evaluating complex SCM tools that forces the evaluator to gain intimate knowledge of a tool to effectively assess the tool's merits for a given software development effort.

THE DEVELOPMENT AND USE OF AN EVALUATION MECHANISM FOR THE ASSESSMENT OF SOFTWARE CONFIGURATION MANAGEMENT TOOLS

I. Introduction

General Issue

In order for an Air Force program to succeed, it is critical that the manager be able to control and account for the system throughout its life cycle. Today's systems are becoming increasingly dependent on software, resulting in more software components to manage, which exponentially increases the difficulty of maintaining configuration control over the system (Forte, 1990:24). This trend, likely to continue as more and more emphasis is placed on software-performing functions traditionally accomplished by hardware, will challenge managers because software, unlike hardware, does not readily lend itself to visual inspection and is very susceptible to change. As a result, software is difficult to control and manage.

The discipline afforded by Software Configuration Management (SCM) provides a means of managing and controlling system software. Proper application of SCM involves 1) identifying a software hierarchy and each software component within that hierarchy (*Software Configuration Identification*), 2) controlling changes to these software components (*Software Configuration Control*), 3) documenting such changes (*Software Configuration Status Accounting*), and 4) auditing the overall software baseline to ensure that the actual software meets the specified requirements and mirrors the documentation (*Software Configuration Auditing*). Proper performance of these SCM activities will guarantee the integrity, correctness and supportability of a system's software, leading to the success of the overall program. Unfortunately, because software routinely comprises a large percentage of the overall system and is in an almost constant state of transformation due to enhancements and defect correction, employing these SCM activities can be a complex and tedious task (Bersoff and Davis, 1991:105).

Such mutability is characteristic of the software of most defense systems in the Air Force and thus is a nightmare for the configuration manager. The problem of managing the multitudes of software versions and releases that develop during the life of a typically large defense system may be most effectively solved through the use of automated SCM tools (Millradt, 1990:6).

Specific Problem

In the Air Force, software systems either are designed and developed organically, or they are contracted out to the defense industry. Regardless of the development environment, it is critical that program management understand both the requirements associated with the four fundamental SCM activities and the methods, or tools, commercially available to automate much of this management endeavor. Managers of organic (in-house) efforts must first be able to articulate the SCM requirements which will govern the life cycle development/support of their system(s), and then procure an appropriate tool to automate and assist them in their SCM responsibilities. It is essential for those who direct contracted development efforts to have knowledge of SCM requirements and an appreciation of the various SCM tools and technologies available which will better enable them to successfully specify SCM requirements to the contractor. When the Air Force assumes responsibility for life cycle management and support of the configured end item, this awareness will improve the chances that it will be satisfied with the SCM system delivered by the contractor. The failure of many software programs is the direct result of either selecting an SCM tool that was either inadequate or exceeded the needs of the program (Millradt, 1990:6). The specific problem this research study addressed is: Typical Air Force management lacks a clear understanding of SCM requirements and how various commercial tools can help an organization meet these requirements.

Research Objectives

The overall objective in this research effort was to provide Air Force management the compiled information necessary to comprehend the requirements which must be satisfied, the activities which must be performed, and the capabilities which should be sought when selecting tools to establish and operate a successful SCM system. In light of this, we established three specific objectives:

1. Establish a standard framework (i.e. requirements, activities, etc.) for a Department of Defense (DoD) SCM system.

2. Develop an evaluation mechanism which can be used to assess SCM tools for the particular needs of an organization.

3. Evaluate a sample of SCM tools using this evaluation mechanism.

Research Questions

The research objectives were achieved by systematically answering the following questions:

1. What SCM requirements or activities are stipulated by published standards and/or guidelines? Answering this question enabled us to partially meet the first objective.

2. Where and how do SCM requirements play a role in the phases of defense system software development? Integrating the answer to this question with those of the first question enabled the definition of a standard functional framework for an SCM system, thus meeting the first objective.

3. What requirements or activities are driven by the four fundamental elements of SCM? Answering this question enabled us to partially meet the second objective.

4. How are tools modeled and developed to address SCM needs? Integrating the answer to this question with those of the first three questions permitted the development of an evaluation mechanism, thus meeting the second objective.

5. What current tools are available to automate SCM responsibilities? Answering this question provided us with a population of tool developers from which to sample and assess commercially available SCM technologies utilizing our evaluation mechanism and, thereby, met the third objective.

Scope of Research

This research effort only addressed those Commercial-Off-The-Shelf (COTS) SCM tools which are currently being used either directly by Air Force software development organizations or by defense contractors developing software for the Air Force. Additionally, in an effort to align this research effort with the DoD's mandate that all new software development be accomplished utilizing the Ada programming language, we only sampled SCM tools that support Ada. Our assessment of each sampled tool was not intended to be a regurgitation of the vendor's brochure and/or user's manual. Instead, we addressed only those aspects of each tool directly impacting its ability to meet the delineated SCM criteria of our evaluation mechanism. Finally, where an SCM tool was part of either an integrated Software Engineering Environment (SEE) or a broader Software Engineering (SE) tool, we assessed only those functions and capabilities which were specific to SCM.

Overview

This research effort provides Air Force managers or, more precisely, Air Force managers, with an evaluation mechanism, or taxonomy, with which to assess potential SCM tools for use on Air Force software development efforts.

Chapter 2 includes the results of an intensive review of published literature concerning the SCM discipline. We reviewed literature pertaining to the four fundamental elements of SCM, the current SCM requirements delineated in defense and industry standards and guidelines, the SCM activities involved in life cycle development and support of defense systems, the fundamental SCM models upon which all current SCM tools are developed, and documented evaluations of various automated tools.

In chapter 3, we describe the methodology we employed to meet the objectives of this research effort. In particular, we define our approach concerning the development of our evaluation mechanism and the performance of the SCM tool selection and evaluation.

In chapter 4, we present our analysis of the criteria compiled to develop the mechanism with which we evaluated each of the tools sampled.

Chapter 5 summarizes our conclusions and recommendations, and also includes a discussion of the strengths and limitations of our SCM tool evaluation mechanism, the resulting tool assessment, and our recommendations for further study.

II. Literature Review

Introduction

As part of this research effort, we performed a search of published information dealing with Software Configuration Management (SCM) and its associated tools. The results of this search are presented in four sections. The first section provides a cursory overview of the SCM discipline based on our review of technical journals. The second section addresses the four fundamental elements of SCM. This part of the review fully develops the theoretical framework behind each fundamental SCM activity. Sources included software management textbooks and technical journals. The third section highlights the generally accepted SCM policy and activities involved in DoD software development and support. This section details our findings based on the review of software management textbooks, educational institution technical briefs, and published standards of the military, DoD, and private industry. The fourth and final section describes the current models around which SCM tools are developed. Sources for this section included software management textbooks and technical journals. This literature review effort enabled us to gain the necessary understanding and appreciation of SCM requirements and tool functionality to develop an evaluation mechanism for the assessment of the SCM tools sampled.

Overview of Software Configuration Management

Configuration management is the "set of techniques used to help define, communicate, and control the evolution of a product or system through concept, development, implementation, and maintenance phases" (Sweetman, 1990:5). For years after its conception, the discipline of configuration management was applied to the development of either hardware systems or hardware elements of hardware-software systems (Bersoff, Henderson, and Siegel, 1980:24). While such development efforts addressed the configuration management of system hardware with meticulous detail, system software was treated as a single entity, whose visibility during the overall system evolution was suppressed (Bersoff, Henderson, and Siegel, 1980:24). However, as hardware became more sophisticated, faster and more powerful, software became more prevalent in the areas of application and percentage of overall cost (McCarthy, 1980:263). With the evolutionary advances of the software industry, management of software development projects has come under increasingly rigorous scrutiny.

Although configuration management was originally created to enable managers to control hardware production, its principles have been tailored and refined to apply to software development, production, and maintenance (McCarthy, 1980:263). According to Bersoff, SCM provides the discipline for identifying the configuration of a software system at discrete points in time, thereby systematically controlling changes to the configuration while maintaining its integrity and traceability throughout the system life cycle (Bersoff, 1980:381). Unfortunately, because SCM is an immature discipline still requiring further study, attempts by managers to implement SCM have sometimes failed (Bersoff, Henderson, and Siegel, 1980:24).

Fundamental Elements of Software Configuration Management

It is commonly accepted among software management academicians and practitioners that software configuration management can be broken down into the following four fundamental activities: (1) software configuration identification; (2) software configuration control; (3) software configuration auditing; and (4) software configuration status accounting. Without a thorough understanding of each of these, an effective analysis in the realm of SCM cannot be performed (Bersoff and Davis, 1991:107).

Software Configuration Identification. In order to track software effectively as it continually changes throughout its life cycle, a manager must be able to clearly define the components which comprise the system (Bersoff, Henderson, and Siegel, 1980:27). This is the goal of software configuration identification, a fundamental element which involves specifying and identifying all software components throughout the life cycle of the system (Mission Critical

Computer Resources Management Guide, [no date]:10-2). Software configuration identification accomplishes this by

1. breaking the system software down into a number of known manageable components,

2. defining each component,

3. uniquely labeling each component, and

4. uniquely labeling the various revisions that appear as these components change over time (Mission Critical Computer Resources Management Guide, [no date]:10-3).

Breaking Down the System Software. The decisions involved in decomposing the system software into components, generically referred to as software configuration items (SCIs), are probably the most important decisions to be made by project management (Berlack, 1992:81). This process is closely intertwined with the specification, analysis, and design of the overall system. The software design hierarchy enables management to identify and control changes to the various SCIs during the system's life cycle, and to estimate manpower and resources required for each SCI's development, while simultaneously tracking its progress (Berlack, 1992:81).

In general, the design hierarchy of the software should be structured so its functionality is easy to pinpoint and change, if necessary (Whitgift, 1991:17). Figure 2-1 illustrates how a software system can be broken down hierarchically into computer software configuration items (CSCIs), CSCIs into computer software components (CSCs), and CSCs into computer software units (CSUs) or into other CSCs which are, in turn, broken into CSUs. In this hierarchy of SCIs, CSUs represent the most elementary units defined. A CSU usually refers to the embodiment of a specific function (an algorithm or, later in the development cycle, the line(s) of software code that implement the algorithm). Regardless of the hierarchical level, each type of element - whether a CSCI, CSC, or CSU - represents an SCI since it is identifiable, controllable, necessary, traceable, functional, modular and homogeneous (Berlack, 1992:84). Along these lines, software configuration identification ensures that a software system will be divided into smaller, less complex and more manageable items (Whitgift, 1991:18).

Defining System Software and Components. The system software is comprised of one or more CSCIs, each of which must be defined. At the very minimum, the software's external functional and performance requirements should be described, along with the design constraints and attributes of the software as a whole (Berlack, 1992:86).

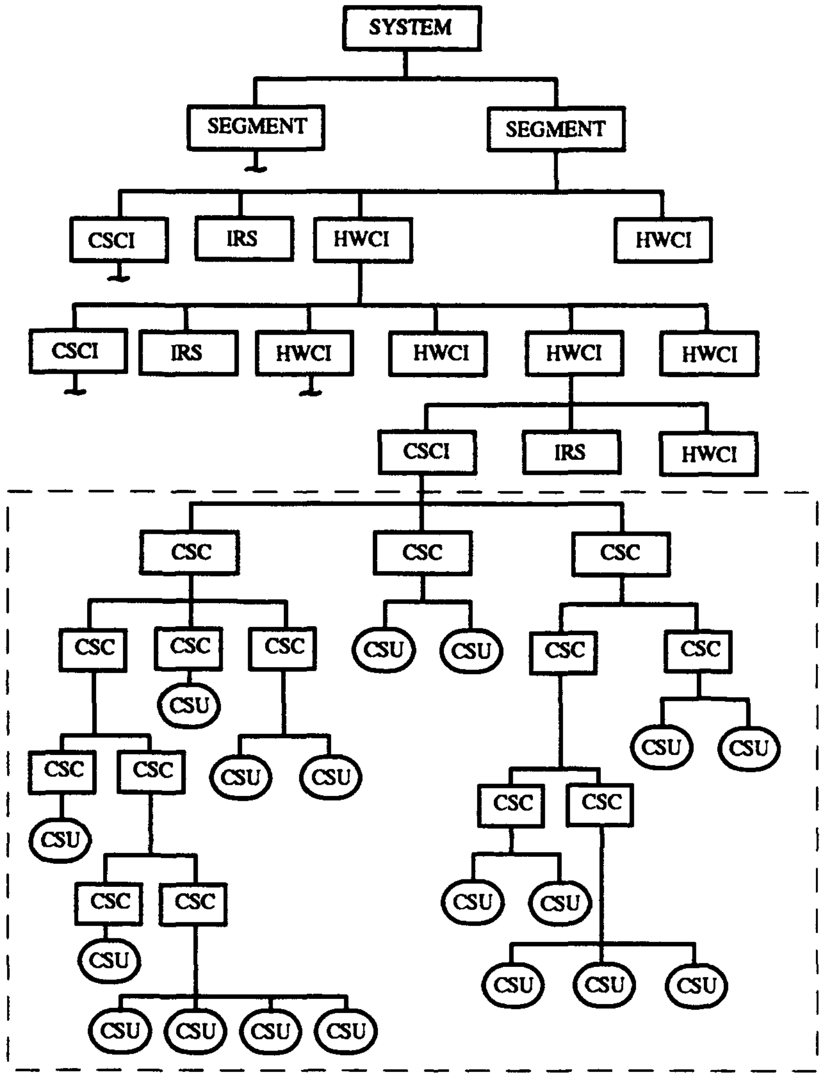

Figure 2-1. Software System Hierarchy (Berlack, 1992:82)

This type of information is recorded in CSCI level documents such as the Software Requirements Specification, Interface Requirements Specification, and Software Design Document. As lower-level software elements are developed, more detailed information is incorporated into these top-level CSCI documents. Data describing how an element was produced/compiled, and when and why changes were made, should be documented for each SCI (Whitgift, 1991:63). This information should be closely coupled with the SCI itself. Descriptive information about an SCI can be included as a cover sheet in the case of a document, or as commented lines in an element of source code (Whitgift, 1991:63). Formally defining the system software and its constituent elements ensures that the status accounting function can effectively obtain information, as needed, concerning the system software.

Labeling SCIs. Each SCI in the design hierarchy must be labeled to provide it with unique identification (Whitgift, 1991:56). Usually, labeling schemes do make use of the design hierarchy wherein each incorporated SCI is named in a manner that it can be identified as a sibling of a certain component and yet be distinguishable from the other components with the same parent (Whitgift, 1991:56). Figure 2-2 illustrates this concept.

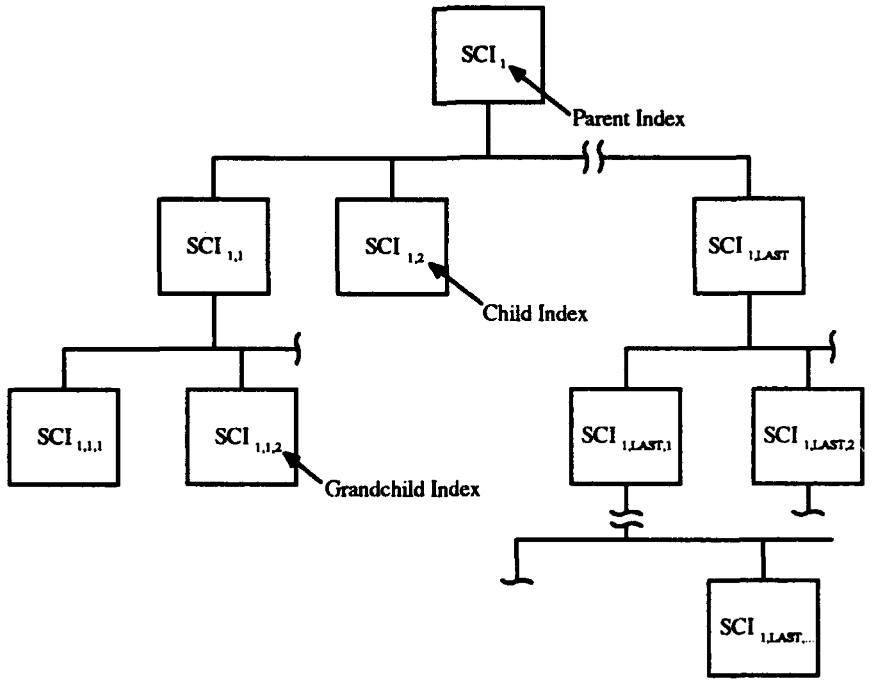

Figure 2-2. Component Labeling Scheme (Bersoff, 1980:111)

As figure 2-2 indicates, the labeling scheme should explicitly exhibit the relationships among the SCIs in the design hierarchy, or tree structure. Figure 2-2 illustrates how indices can be used to label a particular SCI while explicitly showing its parentage (Bersoff, 1980:111). For example, the software configuration item $SCI_{1,2,3}$ is the third offspring of the second offspring of the first software configuration item of the system.

The labeling mechanism must manifestly distinguish between individual SCIs while also clearly identifying the baseline level of each SCI; this particular level refers to the review or approval level of an SCI at a particular point in time during the life cycle of the system (Berlack, 1992:100). In a larger context, a baseline is an SCI, or collection of SCIs, which specifies one or more CSCIs at some "snapshot" in time (Bersoff, Henderson, and Siegel, 1980:27). The need for a clear and efficient labeling scheme becomes even more critical when addressing the issue of identifying SCI versions, which appear as SCIs change over a period of time.

Labeling SCI Revisions. As SCIs evolve through a series of changes, each change must be distinguished from all other changes (Whitgift, 1991:57). There are several different types of revisions: a minor change to an SCI (syntax, spelling, organization, etc.) results in a level revision, while a major change (functionality, interface) results in a release revision, and concurrent changes to an SCI result in variant revisions (Whitgift, 1991:58). This latter revisioning permits an SCI to have multiple configurations, each of which specifies the SCI needed to satisfy a given environment (operating platform, customer requirement, temporary change, etc.). Figure 2-3 illustrates how each type of revision can be identified. In this example, when an SCI is first labeled it is identified as revision 0, whether it is a release or a variant. For instance, 1.0 indicates the first release of an SCI identified, and 1.1 indicates the first revision of that release, whereas, 2.0 indicates the second release and 2.1 is the first revision of the second release. This concept also applies to the identification of variants, both permanent and temporary. The first variant of release 2.0 is 2.0.1.0. The number 1 indicates the first variant and the second number 0 indicates the revision number of that variant. A temporary variant represents a configuration that exists for a short time and eventually is merged with one or more permanent variants of that SCI as illustrated in figure 2-3 below (Whitgift, 1991:38).

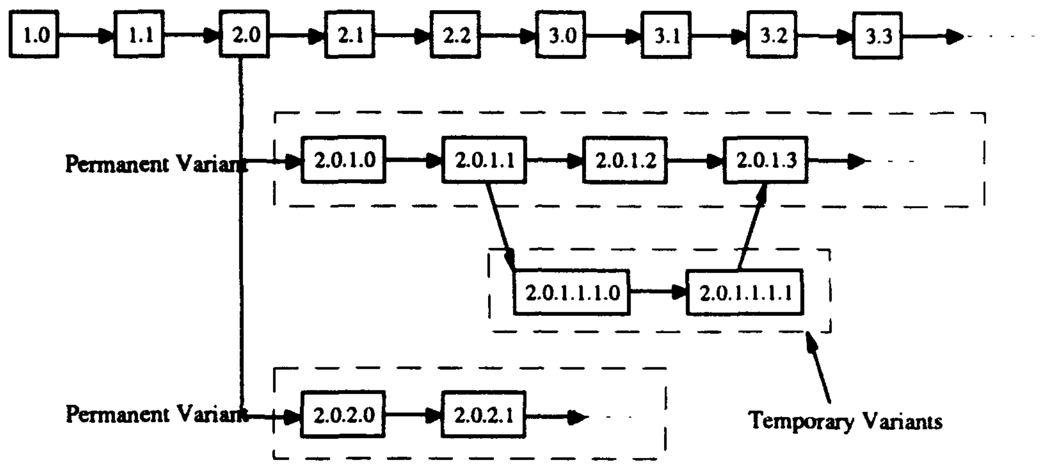

Figure 2-3. Labeling Scheme for Component Revisions (Whitgift, 1991:58-60)

By combining the concepts of SCI labeling and SCI revision labeling, both the genealogy and the version of the software can be identified. For example, the first child of the parent SCI is $SCI_{1,1}$ and the second revision of the first release is 1.2. When these labeling schemes are combined with each other, the second revision of the first release of the first child is $SCI_{1,1}[1.2]$.

If SCIs are clearly identified, their changes and respective baselines can also be identified. This provides an explicit documentation trail that links the stages of the software life cycle (Bersoff, Henderson, and Siegel, 1980:28). In other words, the identification of software configurations will enable a manager to know the system software's stage of development, the history of each software change, and how the SCIs and their changes are interdependent.

Software Configuration Control. Software is highly susceptible to change because of the user's tendency to add new features or capabilities, to correct past errors, or to improve efficiency (McCarthy, 1980:267). The control of software configuration requires the focused management of such changes by accomplishing approval, monitoring, and control of the conversion of design objects into system software configuration items, followed by the changes to these items (Bersoff, Henderson, and Siegel, 1980:221). Bersoff states that effective software configuration control involves certain basic ingredients:

- documentation,
- an organizational body, and
- procedures (Bersoff, 1984:82).

These ingredients should insure that changes are processed, communicated, and incorporated in an orderly manner (Berlack, 1992:109).

Documentation. This process includes both change and baseline documentation. (Bersoff, Henderson, and Siegel, 1980:199). Software is only visible through its documentation (i.e., listings, design specifications, etc.), and is only comprehensible when it is logically organized and controlled as a baseline. Change documentation enables management to formally define and precipitate proposed alterations (Bersoff, Henderson, and Siegel, 1980:199).

Baseline documentation consists of approved documents and related code that specify and implement the software at a point in time in its development (Berlack, 1992:109). A particular baseline documentation should reflect both the evolutionary status (i.e., developmental maturity level) and the revolutionary status (i.e., version level) of the CSCI (Bersoff, Henderson, and Siegel, 1980:199). Controlling this form of documentation enables management to provide an independent, common frame of reference to all observers of the system (Bersoff, Henderson, and Siegel, 1980:180).

Software configuration control requires the documentation of changes, proposed to either enhance capabilities or correct a deficiency (Berlack, 1992:111-113). Change documentation, such as Change Requests, Fault Reports, and Engineering Change Proposals (ECPs), provides a method of tracking a change from its request to its implementation (Whitgift, 1991:131). The choice of form depends on the circumstances necessitating the change. While Change Request forms are designed to describe how the software must be changed or enhanced, Fault Report forms document the symptoms of a fault, anomaly, or bug (Whitgift, 1991:139). Both the Change Request and

Fault Report forms are intended to address changes to non-baselined items. Engineers will continually lobby for changes to be categorized as enhancements and therefore will employ Change Requests, while the user will lobby for changes to be classified as defect corrections and thus will utilize the Fault Report (Berlack, 1992:113). Both Change Requests and Fault Reports can be precipitators of ECPs, depending on whether or not either results in a change to baselined software or its documentation (Bersoff, Henderson, and Siegel, 1980:200). An ECP delineates both the changes which are to be made to baselined components and the resulting cost, schedule, and performance impact; therefore, it constitutes an amendment to the contract between the developer and user (Berlack, 1992:133).

Organizational Body. Proposed changes must be reviewed and then either approved or disapproved. The organizational authority for such decisions resides with the Configuration Control Board (CCB) (Bersoff and Davis, 1991:106). The CCB is the heart of the configuration control function and, as such, must be organized to be responsive to project requirements (Bersoff, Henderson, and Siegel, 1980:187). The CCB evaluates changes based on such areas as operational impact, classification, interface impact, cost impact, schedule impact, feasibility, and impact to quality and reliability (Berlack, 1992:143). Because of the range of its responsibilities, the CCB should be comprised of representatives of every operational phase having a legitimate interest in the proposed change (Whitgift, 1991:136). CCB representation normally includes, but is not limited to, program management, system and software engineering, software configuration management, quality assurance, and integrated logistics support (Berlack, 1992:141). During busy times, the CCB relies on and delegates authority to review and screening boards. Such boards can save many hours of CCB time by sorting through changes, thereby enabling the CCB to concentrate on complete and workable change proposals (Berlack, 1992:144).

Procedures. Software configuration control provides for procedures by which proposed changes can be reviewed, evaluated, and implemented (Whitgift, 1991:153). According to Bersoff, Henderson, and Siegel, "Procedures form a logical and enforceable series of steps by which changes (both evolutionary and revolutionary) to the system are processed" (Bersoff, Henderson, and Siegel, 1980:181). Figure 2-4 depicts the basic steps involved in the change control process. Notice that regardless of whether or not a proposed change is approved, management must establish procedures for - and monitor the archiving of - disapproved changes for future reference.

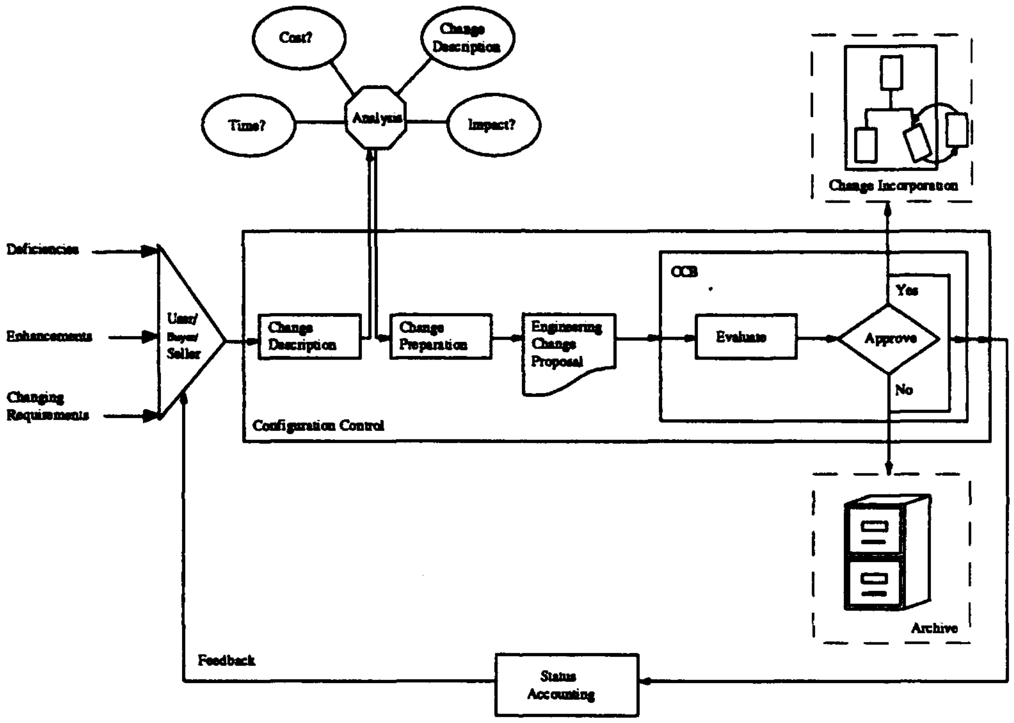

Figure 2-4. Generic Change Control Process (Bersoff, 1984:84)

It is noteworthy that, even though actual change incorporation is not an SCM function, monitoring any change implementation process which results in change incorporation does constitute an SCM function (Bersoff, Henderson, and Siegel, 1980:29). Management must control the procedures by which developers create new code, modify or delete existing code, and share code with other programmers (Babich, 1986:84). This is accomplished by ensuring that the software library, or repository of code and documentation, is secure and protected against unauthorized access (Whitgift, 1991:181). Read access must be controlled to prevent unauthorized disclosure, while write access must be controlled to prevent unauthorized change or deletion (Whitgift, 1991:181). At the same time, because the software repository is shared among the members of the development team, some sort of shared access must be coordinated (Babich, 1986:68).

Although configuration control appears to divert or inhibit developers from doing their job, this situation should not and need not occur. If properly implemented, configuration control is sensitive to the context in which developers must operate; this procedure applies the right degree of constraint to ensure that developers work in a responsible and disciplined way while contributing to the objectives of the team (Whitgift, 1991:125). It should be noted that configuration control does not just "go away" when production is completed and the contractor turns the system over to the government for operational use; such control must extend as the government continues to makes its own changes to the delivered item(s) (Dean, 1979:25). Therefore, software configuration control enables a manager to ensure that his or her software system consists only of authorized software and the authorized changes.

Software Configuration Auditing. This procedure provides mechanisms for determining the degree to which a particular software configuration mirrors the software configuration represented in baseline and requirements documentation, and for establishing or sanctioning baselines (Bersoff, 1980:386). The two most prevalent or recognized types of configuration audits are the "functional" and "physical" varieties, which occur at the conclusion of software development. The functional configuration audit validates the performance of one or more CSCIs in satisfying the user requirements, whereas the physical configuration audit verifies that the software implementation is accurately and adequately reflected in its documentation (Berlack, 1992:176,178). Once both audits are successfully completed, the product baseline is formally established. However, the SCM function of auditing must be conducted throughout the software development life cycle. Not only should the final developmental configuration be audited, so also should each configuration leading to the establishment of the functional, allocated, design, and operational baselines (Bersoff, Henderson, and Siegel, 1980:31). Regardless of the particular time frame in the life cycle, configuration auditing serves two purposes: configuration verification and configuration validation (Bersoff, 1980:386). Configuration auditing verifies that identified software configurations are what they were intended to be, and it validates that they fulfill the functions corresponding to the respective milestone points (Bersoff, Henderson, and Siegel, 1980:232). In this manner, as software life cycle products are audited and baselines are established, requirements can be traced from baseline to baseline (Bersoff, 1980:386). Quite simply, the success or failure of software configuration auditing reflects the integrity of the system's software.

Software Configuration Status Accounting. The activities involved with the fundamental SCM activities of configuration identification, configuration control and configuration auditing result in a massive amount of data for management to assimilate (Bersoff, Henderson, and Siegel, 1980:285). This is even more the case when one considers that software changes almost continuously. Rarely will a software baseline exist that does not have additional changes. The goal of software configuration status accounting is to create and maintain records of all software baselines, the SCIs associated with each baseline, and the corresponding changes (Bersoff and Davis, 1991:107). Status accounting provides management with the tools by which such software information can be organized to produce a useful and coherent system picture (Bersoff, Henderson, and Siegel, 1980:285). According to Berlack, configuration status accounting ". . . keeps track of the current configuration identification documents, the current configuration of the delivered software, the status of changes being reviewed, and the status of the implementation of approved changes" (Berlack, 1992:153). Status accounting consists of three processes: recording, storing, and reporting (Bersoff, Henderson, and Siegel, 1980:291).

Recording. The purpose of recording is to capture all events and information of significance concerning the development of the software code and documentation (Bersoff, Henderson, and Siegel, 1980:295). Examples of the type of information which should be recorded include development status of configuration items, review status of change requests, item versions and implementation dates associated with any software change activity, differences between multiple versions of an item, number of faults detected in each item and cause of problem reports (Whitgift, 1991:152). As with all other SCM activities, the magnitude of the recording function will depend upon the scope and complexity of the system undergoing development (Bersoff, Henderson, and Siegel, 1980:294).

Storing. The role of storing is to provide a complete and organized data base of all information recorded and of all software documentation and code developed. The platform or foundation upon which the storing process is based is the software development file (SDF), which evolved from the unit development folder (UDF) (Berlack, 1992:154). According to Ingrassia, the UDF is ". . . a structured mechanism for organizing and collecting software development products (requirements, design, code, test plans/data) as they become available" (Ingrassia, 1987:405). Whereas the UDF has addressed data collection at the unit level (i.e., CSU level), the SDF compiles or organizes these abstractions of data into component level (i.e., CSC level) and end item level (i.e., CSCI level) files. In a data base or automated environment, a file system with corresponding directory structure is established, enabling the software system to be broken down into subsystems, subsystems into programs, programs into components and components into units. Using this approach, at any point in time data can be stored at the appropriate system abstraction level, new data can be quickly imported and data can be exported to build a report. This also facilitates the regeneration, if necessary, of any or all baselines and the changes to any particular baseline (Bersoff and Davis, 1991:107).

Reporting. The purpose of reporting is to make both software and its developmental history visible to all project participants (Whitgift, 1991:151). This is accomplished by preparing and distributing reports to project personnel which contain necessary day-to-day information concerning the status of the system software development (Berlack, 1992:154). According to Bersoff, Henderson, and Siegel, "These reports, based on the data recorded and stored, will be both scheduled (by the SCM Plan) and ad hoc (in response to inquiries by project participants)" (Bersoff, Henderson, and Siegel, 1980:303). Examples of scheduled reports include CCB meeting minutes, baseline status reports, executive summaries of SCM activities, change request status reports, fault report status, and baseline release notes (Bersoff, Henderson, and Siegel, 1980:299). Each of these reports provides management and/or developers with information at established points in time during the software development life cycle. However, in order to satisfy the myriad of SCM-related questions which project participants pose on a daily basis, the status accounting system must respond rapidly to a variety of queries that may include:

- What is the status of an item?
- Has a change request been approved or rejected by the CCB?
- Which version of an item implements an approved change request?
- What is different about a new version of a system?
- How many faults are detected each month and how many are fixed?
- What is the cause of fault reports (Whitgift, 1991:151-152)?

The completeness of the data recorded, and the thoroughness and discipline used in storing and cataloging the data, will determine how successfully the status accounting system can generate ad hoc reports to satisfy such queries (Bersoff, Henderson, and Siegel, 1980:302).

Software configuration status accounting permits management to trace the history of a software system's life cycle, re-create any past software configuration, and validate any software baseline against a level of documentation (Bersoff, Henderson, and Siegel, 1980:29). It can also provide management with useful statistics upon which to base future costing and scheduling decisions (Berlack, 1992:169). As such, status accounting truly is the capstone to the other fundamental SCM activities, providing a monitoring system to keep the established system of documentation and change control up to date, and it assures that the software being developed will be maintainable (Berlack, 1992:173). However, Berlack pointed out that a status accounting system is only as good as the information within it and its design requires thought and planning to ensure that the data are available and can be updated with available resources (Berlack, 1992:173).

Software Configuration Management in DoD Software Development

Since this research effort focused on DoD software development programs, various standards were reviewed to understand how the DoD currently addresses SCM. These included DoD-STD-2167A, *Defense System Software Development*, DoD-STD-2168, *Defense System Software Quality Program*, and MIL-STD-973, *Configuration Management*. The combination of these standards provides the means for establishing, evaluating, and maintaining quality in software and associated documentation, and are applicable throughout the system life cycle (DoD-STD-2167A, 1988:iii). Additionally, to establish a comparison, we also reviewed IEEE-STD-1042, *Guide to Software Configuration Management*, and IEEE-STD-828, *Software Configuration Management Plans*. Although the DoD is planning to supersede DoD-STD-2167A with DoD-STD-498, *Software Development and Documentation*, this replacement was still in draft form at the time of this writing. Thus, the current approved standard governed. The following paragraphs present a compilation of the SCM activities from the referenced standards that are applicable to all or specific phases of the software system life cycle.

Software Configuration Management Requirements. MIL-STD-973 specifies the configuration management requirements, both for hardware and software, that may be applied to a DoD development program. Only the software requirements were addressed for this research effort. This particular standard was intended to be the only DoD document for configuration management requirements, superseding previous configuration management standards, such as MIL-STDs-480, -481, -482, and -483 (MIL-STD-973, 1992:120). The general SCM requirements, as identified in MIL-STD-973, address planning, the four elements of SCM, and data transfer, distribution, and access. Specific requirements for each of the four elements of SCM are identified and discussed in the following paragraphs.

Establish and Document Policies. MIL-STD-973 first identifies the requirements by which a contractor will establish and implement configuration management policies and administration, then specific guidelines are given which the contractor will follow in creating new configuration management procedures, or restructuring existing ones. The end product of this effort is a configuration management (CM) plan whose content and format is governed by the Data Item Description (DID) DI-CMAN-80858A, *Contractor's Configuration Management Plan*. The contractor will plan how to meet the data storage, distribution, access, security, maintenance, and processing requirements of the contract (MIL-STD-973, 1992:19). These stipulations may require traditional hard copies of data or an interactive digital data processing system, such as Computer-aided Acquisition and Logistic Support (CALS). The plans for conducting SCM are included as part of this overall CM plan. Alternatively, as discussed later, they can be included as part of DID DI-MCCR-80030A, *Software Development Plan*, under DoD-STD-2167A.

Identification. For configuration identification, the contractor is required to "incrementally establish and maintain a definitive basis for control and status accounting for a configuration item (CI) throughout its life cycle" (MIL-STD-973, 1992:25). Generally, the contractor will meet this requirement by identifying the SCIs and their configuration documentation; establishing a developmental configuration; establishing functional, allocated, and product baselines for each SCI; defining, documenting, and managing interfaces; establishing an engineering release system for configuration documentation and source code; and assigning identifiers to each SCI, its component parts, and documentation.

Control. As part of configuration control, MIL-STD-973 requires the contractor to regulate changes to all SCIs. This is accomplished by systematically documenting and evaluating proposed changes. Documenting includes describing, justifying, and coordinating a proposed change. Evaluation involves determining the impact of the proposed change on the rest of the system and the approval or disapproval of the proposed change. In this section of MIL-

STD-973 the requirements for the classification of various engineering changes are discussed in detail, including the specific DoD forms and procedures for each change. These change classifications include Engineering Change Proposals, Request for Deviations, Request for Waivers, Specification Change Notices, and Notices of Revision.

Auditing. MIL-STD-973 specifies two configuration audits, the Functional Configuration Audit (FCA) and the Physical Configuration Audit (PCA). The requirements, procedures, and responsibilities are delineated and discussed. For each audit, the contractor is required to develop an audit plan and agenda, and to provide all the necessary information, personnel, and support to conduct each audit.

Status Accounting. In order to perform configuration status accounting, MIL-STD-973 requires the contractor to establish an information management system, as defined in Appendix H of the standard. As part of this configuration status accounting system, the contractor will maintain a complete historical record of all required information. Appendix I of MIL-STD-973 provides guidance for identifying and defining data elements for use in status accounting records and reports. The contractor is required to analyze status accounting data to detect trends in reported problems and to verify that corrective actions have resolved these adverse trends (MIL-STD-973, 1992:64).

MIL-STD-973 applies to all the acquisition phases: Demonstration and Validation, Engineering and Manufacturing Development, Production and Deployment, and Operation and Support. Although the standard does not specifically address acquisition phases in the requirements sections, its final section presents methods for tailoring MIL-STD-973 to the requirements of specific acquisition phases. In contrast, the SCM requirements in DoD-STD-2167A depend on the software development activities that occur during the acquisition phases.

Software Life Cycle. The system life cycle is partitioned into acquisition phases, each defining a stage of the development of the system. For instance, DoD-STD-2167A describes the development of software systems as activities within these acquisition phases. While software-unique activities normally occur during the phase of Engineering and Manufacturing Development (EMD), these same activities can also occur during Demonstration and Validation, when developing a prototype, or during Production and Deployment or Operations and Support, when modifications are required. For ease of discussion, the software development activities are presented in line with the overview of the DoD software development life cycle shown in Figure 2-5. Although it appears DoD-STD-2167A specifies the use of the Waterfall method for software development, any development method can in fact be used, depending upon the contract requirements (DoD-STD-2167A, 1988:iii/iv). Thus, because software activities are not regimented to occur only during EMD, they can also occur during any of the phases shown in Figure 2-5.

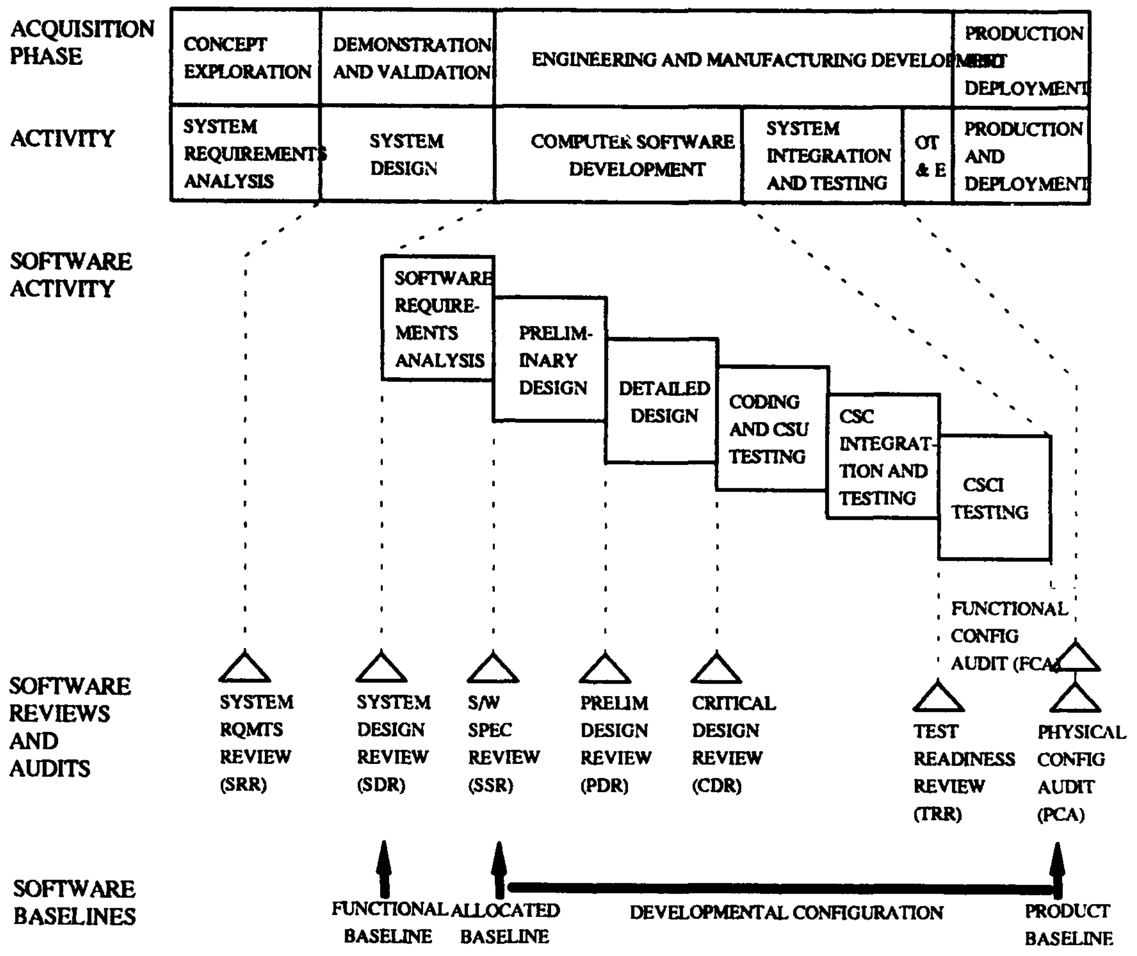
ACQUISITION PHASE
CONCEPT EXPLORATION
DEMONSTRATION AND VALIDATION
ENGINEERING AND MANUFACTURING DEVELOPMENT
PRODUCTION AND DEPLOYMENT
ACTIVITY
SYSTEM REQUIREMENTS ANALYSIS
SYSTEM DESIGN
COMPUTER SOFTWARE DEVELOPMENT
SYSTEM INTEGRATION AND TESTING
OT & E
PRODUCTION AND DEPLOYMENT
SOFTWARE ACTIVITY
SOFTWARE REQUIRE-MENTS ANALYSIS
PRELIM-INARY DESIGN
DETAILED DESIGN
CODING AND CSU TESTING
CSC INTEGRAT-TION AND TESTING
CSCI TESTING
SOFTWARE REVIEWS AND AUDITS
SYSTEM RQMTS REVIEW (SRR)
SYSTEM DESIGN REVIEW (SDR)
S/W SPEC REVIEW (SSR)
PRELIM DESIGN REVIEW (PDR)
CRITICAL DESIGN REVIEW (CDR)
TEST READINESS REVIEW (TRR)
FUNCTIONAL CONFIG AUDIT (FCA)
PHYSICAL CONFIG AUDIT (PCA)
SOFTWARE BASELINES
FUNCTIONAL BASELINE
ALLOCATED BASELINE
DEVELOPMENTAL CONFIGURATION
PRODUCT BASELINE

Figure 2-5. Computer Software Development Cycle (Berlack, 1992:47)

In each of these phases, specific SCM activities must occur. DoD-STD-2167A specifies the SCM requirements by the software development activities shown above. The requirements specified by both DoD-STD-2167A and DoD-STD-2168 are presented in the following paragraphs under the software development activities in which they occur.

System Requirements Analysis and Design. During these activities, the contractor develops a plan which specifies the SCM organization, procedures, and schedules. The government can receive a copy of the contractor's SCM plan either by requiring it as part of a Software Development Plan (SDP) or as part of a system Configuration Management Plan (DoD-STD-973, 1992:18-19; DI-MCCR-80030A, 1988: 1-2). In conjunction with either of these options, the contractor develops a Software Quality Program Plan (SQPP), which documents the contractor's procedures for implementing a software quality program (DoD-STD-2168, 1988:2). A preliminary set of system requirements is then defined and documented in a preliminary System Specification. Status accounting records and stores the preliminary versions of the SDP, SQPP, and System Specification, thus beginning the developmental history of the program. A System Requirements Review is conducted to review, evaluate, and establish the formal requirements for the system, some of which will later filter down to the software. The product of this review is a preliminary System Specification which, upon Government acceptance, is placed under contractor configuration control. The system is then partitioned into two types of Configuration Items (CIs), Hardware Configuration Items (HWCIs) and Computer Software Configuration Items (CSCIs), and all are labeled. The Air Force Materiel Command uses Computer Program Identification Numbers (CPINs) to identify the CSCIs in its systems; however, there is no standard DoD software identifier system (Ferens, 1993). The system requirements are allocated among the HWCIs, CSCIs, and manual operations, and then documented in a System/Segment Design Document (DoD-STD-2167A, 1988:19). Also, during these activities, the configuration baselines (i.e., functional, allocated, and product) and the associated documentation are identified and labeled. Next, the contractor defines preliminary sets of software and interface requirements and documents them in a preliminary Software Requirements Specification (SRS) and a preliminary Interface Requirements Specification (IRS), respectively, for each CSCI, after which they are placed under contractor configuration control. The Functional Baseline (first of a series) is established after the successful completion of a System Design Review (SDR); it represents both the system's approved and documented operational characteristics and its design constraints (MIL-STD-973, 1992:12). The System Specification, System/Segment Design Document, SDP and SQPP are all finalized. Once accepted by the government, they are placed under government configuration control. After the successful completion of SDR, software design and development begins, involving software requirements analysis, preliminary design, detailed design, coding and unit testing, component integration and testing, and CSCI level testing (Berlack, 1992:52).

Software Requirements Analysis. During this activity, the preliminary software and interface requirements are formalized in a finalized SRS and IRS. These two documents receive government approval or disapproval at a Software Specification Review (SSR). The purpose of this review is to ensure that all the requirements specified for software during the System Design activity have actually been allocated to a CSCI and that specified and derived requirements are adequately defined in the IRS and SRS. Upon successful completion of this review and acceptance of the SRS and IRS, the Allocated Baseline is established and these documents are placed under government configuration control.

Preliminary Design. The Preliminary Design activity signifies the establishment of the Developmental Configuration. This is a contractor-controlled internal baseline which describes the evolving configuration of the software being developed. The preliminary design for each CSCI is developed, requirements are allocated from the SRS and the IRS to the Computer Software Components (CSCs) of each CSCI, and the design requirements are established for each CSC (DoD-STD-2167A, 1988:23). This information is captured in a preliminary Software Design Document (SDD). In addition, the interface designs for each CSCI are developed and documented in a preliminary Interface Design Document (IDD) for each CSCI. Plans and test requirements for integration and testing each CSC are developed and documented in a Software Test Plan (STP) for each CSCI. These three documents must be based on, and either directly traceable back to or derived from, the requirements specified in the SRS and IRS. Software Development Files (SDFs) are established for all CSCIs and CSCs. The contractor may choose either to logically group CSCs into single SDFs or to establish a separate SDF for each CSC (DoD-STD-2167A, 1988:14). SCM will support one or more Preliminary Design Reviews (PDRs) to evaluate the initial design and test plan for each CSCI. After the contractor receives government approval for the preliminary design and documentation, the contractor's Developmental Configuration for each CSCI is established. The SDD, IDD, STP, and CSC test requirements are incorporated into the Developmental Configuration under contractor configuration control.

Detailed Design. After the initial designs are established, the Detailed Design activity begins. In this activity, a more detailed and final design is developed for each CSCI. Requirements are allocated from the CSCs to the Computer Software Units (CSUs) of each CSCI and design requirements are established for each CSU (DoD-STD-2167A, 1988:25). The SDD and the IDD are updated to include the detailed design information. In this phase, SDFs are established for each CSU, or logical grouping of CSUs, similar to the SDFs for the CSCs. Test requirements, responsibilities, and test cases for each CSC and CSU are recorded in the respective software development files. Test cases are documented in a Software Test Description (STD) for each CSCI. SCM will support a Critical Design Review (CDR) for one or more CSCIs to review the design and test documents and to evaluate the proposed detailed design for each CSCI. As in the PDR, once approval is received from the Government, the detailed SDD, IDD and STD for each CSCI are incorporated into the developmental configuration, then placed under contractor configuration control.

Coding and CSU Testing. In the Coding and CSU Testing activity that follows CDR, SCM places the updated SDD and software source code listings for each successfully tested

CSU under contractor configuration control, as it is incorporated into the Developmental Configuration. Internal design and code walkthroughs are performed to review design and coding efforts for correctness and compliance to the contractor's internal coding standards. It is critical that status accounting maintains a record of all action taken for each walkthrough in the SDFs (Berlack, 1992: 56), which are updated to include source code, test procedures and reports, and design documentation.

CSC Integration and Testing. This integration and testing activity involves SCM functions similar to Coding and CSU Testing, the difference is that this activity involves integrating the CSUs into CSCs. Testing and evaluation are performed at the CSC level. The SDD is updated, and the source code listings and test results are added to the respective SDFs. SCM supports a Test Readiness Review (TRR) by providing CSC test reports and the latest versions of the design documents and source code listings for each CSC. As each test case is identified in the STD, formal test procedures are developed and documented in it. The updated STD and source code for each successfully tested CSC are incorporated into the Developmental Configuration, then placed under contractor configuration control (DoD-STD-2167A, 1988: 29).

CSCI Testing. In this activity of testing, the CSCs are integrated into their respective CSCIs and tested, using the formal procedures in the appropriate STD. SCM is responsible for identifying the exact version of the software for each CSCI in a Version Description Document (VDD). SCM will support a Functional Configuration Audit (FCA) and a Physical Configuration Audit (PCA) at the end of this activity. According to MIL-STD-973, this support involves audit preparation, assistance during the audit and post audit actions (e.g., publishing audit minutes and recording audit results in a configuration status accounting system). These audits are intended to verify that the software meets the specified performance requirements and that the software accurately reflects its documentation or revised documentation reflects its software. When the FCA and PCA are successfully completed, the Developmental Configuration for each CSCI becomes the Product Baseline, at which time the CSCI configuration is under government control. The final SDD, IDD and source code listings become the Software Product Specification (SPS) for each CSCI. Once the SPS, VDD and users manuals are evaluated and approved, they are placed under government configuration control. Depending on the system, SCM may support System Integration and Testing when the system hardware is integrated with the CSCI(s). With government approval, the contractor will prepare any necessary changes to baselined documentation and source code resulting from System Integration and Testing.

Production and Deployment. During the Production and Deployment phase, SCM supports software maintenance resulting from problem/change reports. Proper configuration control and status accounting must be performed to ensure that only approved changes are made to the affected baselines, and that they are properly recorded. SCM also supports the development of enl ıcements, modifications, or block changes (group of modifications) similar to the previous life cycle phases.

In conclusion, MIL-STD-973 addresses many SCM functions related to the four fundamental SCM elements which do not fit precisely into any one or two software development activities. In fact, according to the tailoring guide for MIL-STD-973, most of the requirements apply to all acquisition phases and software development activities (MIL-STD-973, 1992:102-106). MIL-STD-973 primarily deals with the details of "what" and "how" configuration management is to be accomplished, whereas DoD-STD-2167A specifies the "what" and "when" for SCM. DoD-STD-2168 parallels MIL-STD-973 in the activities of configuration control and auditing by specifying internal contractor control and audits to be performed on the contractor's own SCM efforts. The comparison and intertwining of the standards proved to be a difficult task in that many of the standards referenced by DoD-STD-2167A had been superseded by MIL-STD-973, and any direct correlation was masked. DoD-STD-2167A and MIL-STD-973 contain some overlap since the intent of MIL-STD-973 was to collate all CM, including SCM, under one standard; however, DoD-STD-2167A has not yet been replaced.

Models used in Developing SCM Tools

This section presents the models upon which most current SCM automated tools are based. Specifically, the Checkout/Checkin, Composition, Long Transac~~n, and Change Set models are addressed. According to Feiler, these ". . . four models have been derived from examining a number of commercial systems providing CM functionality, be it CM tools, multi-user CASE tools, or environment frameworks with CM capabilities" (Feiler, 1991:45). The four models can be differentiated by the manner in which each defines a working context for changing the product, maintaining the product's version history, supporting and managing concurrent modifications, and managing and propagating logical changes (Feiler, 1991:2). In turn, these are discussed for each model.

Checkout/Checkin Model. SCM systems based on this type of model consist of a repository tool and a build tool (Feiler, 1991:5). The repository tool provides management with the mechanism to control the creation of new versions of files and then to store the multiple resultant versions (Feiler, 1991:5). Given a description of the components that make up a product, the build tool provides for automatic generation of derived files such as object code and linked executables (Feiler, 1991:5). Using these two tools, the Checkout/Checkin model targets two key features: maintaining version history of individual files and controlling concurrent modification of files (Feiler, 1991:6).

Definition of Working Context. The developer does not modify components in the repository. As the model's name implies, a component must first be checked out, or copied, to a directory in the file system. Therefore, the file system constitutes the general work area where components can be viewed and/or modified depending upon the access rights applied (Feiler, 1991:10); SCM systems based on the Checkout/Checkin model do not control access to components outside the repository. Instead, management must establish work areas for its developers by creating sub directories with appropriate access rights within the file system (Feiler, 1991:10).

Maintenance of Version History. The repository's evolving system files are stored in a directory hierarchy that mirrors that system's actual structure. The repository tool ensures that when files are checked out of the repository, modified, and then checked back, they are stored as a new version (Feiler, 1991:5). The Checkout/Checkin model's file versioning capability permits the creation of sequential revisions, temporary or developmental branching, and permanent variant branching of files that need historical tracking. Additionally, as Figure 2-6 shows, two versions from different branches can be merged into one new version in a single branch.

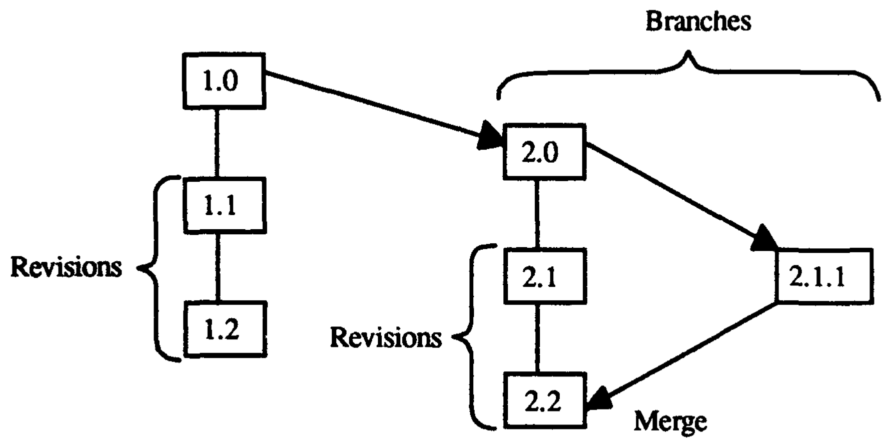

Figure 2-6. File Versioning in the Checkout/Checkin Model (Feiler, 1991:7)

Control of Concurrent Changes. Incorporating a software change routinely involves modifying more than just one file or document and is often accomplished concurrently with other changes. At any given time, multiple programmers may need to modify a given file for various reasons. The Checkout/Checkin model controls this concurrent change activity to maintain the integrity of each component and system configuration. The retrieval of files can be controlled so that only one person at a time can check out a file from the repository. Alternatively, file versions can be checked-out as the initial version of a new branch so that, when all file versions predicated on the same initial file version have been checked back into the repository, they are merged together into one file version (Feiler, 1991:9).

Management and Propagation of Change. A logical change to the overall system software often requires several components to be modified together. Unfortunately, even though all the necessary components can be checked out to one work area, modified, and checked back into the repository as new component versions, the repository has no way of automatically knowing that these components involve logically related changes (Feiler, 1991:11). Component versions can only be identified as part of a logical change through manual means. The label for each applicable component can reference the governing logical change, or comments can be included within each of the applicable component files indicating the logical change that drove the modification (Feiler, 1991:11). Because the Checkout/Checkin model is predicated on the versioning of individual files, rather than variant configurations of files, propagation of changes (to other system variants) has no real meaning in this model.

Composition Model. Whereas the Checkout/Checkin model focuses on supporting the evolution of components, the Composition model focuses on supporting the creation and evolution of system configurations (Feiler, 1991:15). A configuration consists of a system model and version selection rules. The system model is a listing of all the components that make up that system. The version selection rules are the necessary search path or predicate logic options used to indicate which version of each system component should be chosen to build a particular configuration and to which location(s) each component should be checked out (Feiler, 1991:15).

Definition of Working Context. In the Composition model, a configuration serves as the working context. When developers need to modify the system software they either access an existing system configuration in the repository, or they define and create a new configuration by choosing the combination of appropriate selection rules and system model that defines the desired configuration (Feiler, 1991:18). The selection options can point to components in other developers' work areas, system variants, and labeled branches representing particular development paths (Feiler, 1991:17). The applicable component versions which comprise that configuration are then automatically made available in (i.e. copied to) the file system.

Maintenance of Version History. Despite emphasizing configurations, the Composition model still evolves a system by versioning individual components (Feiler, 1991:19). The system models and selection rules that comprise configurations are treated as components and are stored as versions in the repository. Each combination of a system model and selection rule is then recognized as a configuration and given a name that reflects the configuration's version history (Feiler, 1991:19). Thus, the version history of system models, selection rules, and configurations are all maintained together in the repository.

Control of Concurrent Changes. As with the Checkout/Checkin model, the Composition model relies on locking and branching mechanisms for concurrency control. The use of appropriate access rights can prevent developers from accessing and changing components in each other's work areas (Feiler, 1991:20). At the same time, developers can cooperate and share components. They can do this by either sharing a work area or including more than one work area in the selection rules, thereby providing availability for the same component(s) in more than one work area (Feiler, 1991:20).

Management and Propagation of Change. Because this model forces developers to use configurations as the working context, logical changes can be identified and managed (Feiler, 1991:21). When a change request is implemented in a particular configuration, all the component versions associated with the logical change are easily identified and retrieved by accessing the version selection rules for that configuration. Unfortunately, while logical changes can be managed they cannot be included in other configurations (Feiler, 1991:21).

Long Transaction Model. SCM systems based on the Long Transaction model focus on supporting the evolution of a whole system as a series of changes made by a team of developers. Changes are performed as transactions having durations of hours, days, or months. Changes either represent the work of one developer or, through a series of nested transactions, the work of a group of developers. Systems based on this model not only manage the repository; they also support the developers while they modify the software within their work areas (Feiler, 1991:23-24).

Definition of Working Context. With SCM tools based on the Long Transaction model, the developer's work area is represented by the workspace concept. Wherein the workspace provides local memory and replaces the use of the file system as the work area (Feiler, 1991:24). A workspace can originate from either the repository or an enclosing workspace (see Figure 2-7). Once originated, a workspace will exist indefinitely as both a working configuration and a series of preserved configurations (Feiler, 1991:25). When a change transaction is finally committed and a new configuration version is created in either the repository or enclosing workspace, the subject workspace can be deleted or used for yet further development (Feiler, 1991:26).

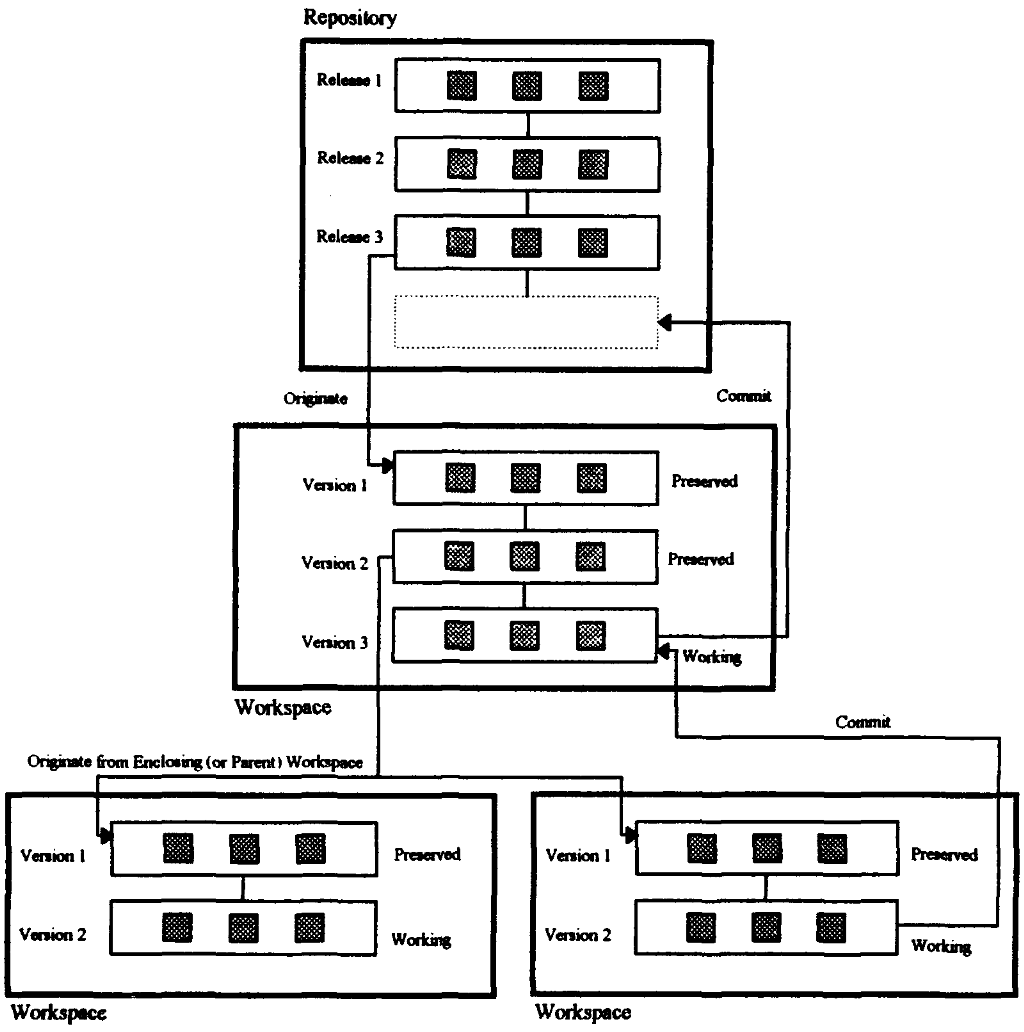

Figure 2-7. Relationship between Repository and Workspace (Feiler, 1991:25)

Figure 2-7 demonstrates the concepts involved in originating a workspace from a particular repository release, originating a workspace from a parent workspace, preserving local configurations within the workspace, then finally, committing a workspace configuration version back into either an enclosing workspace or the repository.

Maintenance of Version History. The configuration versions within the repository represent the version history of the entire system software. These bounded versions result when change transactions from offspring workspaces are committed to the repository, while at the same time, developmental version history is maintained within the offspring workspaces. The workspace concept provides for local version history in that changes can be preserved as a series of immutable configurations within the workspace before they are sufficiently ready to be committed as a new configuration version to the repository or enclosing workspace (Feiler, 1991:24). These preserved configurations also provide checkpoints to which developers can revert.

Control of Concurrent Changes. This model provides a myriad of ways in which concurrent changes can be controlled within one workspace or between two or more workspaces. Concurrency within one workspace can be managed by limiting access to the workspace and/or the preserved and working versions within the workspace (Feiler, 1991:30). Concurrency between workspaces can be managed by controlling access to individual components within workspace versions, limiting the number of offspring workspaces to one, and allowing only one workspace per subsystem (Feiler, 1991:30). Concurrency between sibling workspaces can also be managed by allowing the first workspace finished to successfully commit a change, forcing concurrent workspaces to commit their changes later, identifying any conflicts resulting from merging these changes (Feiler, 1991:31).

Management and Propagation of Change. While this model provides no schemes for managing groups of logically related changes, it does make the propagation of change possible. Such changes are propagated whenever a transaction is committed and the repository or enclosing workspace is updated with a new configuration, and whenever a developer originates or updates his or her workspace with the most current configuration version (Feiler, 1991:34).

Change Set Model. Unlike the first three models which focused on managing the versioning of components and/or configurations, the Change Set model focuses on managing logical changes to system configurations. In this model, each configuration consists of a baseline and a set of change sets. In the context of a component, a change set is the set of differences between the two component versions, whereas in the context of a configuration, a change set is the collection of differences of those components that have been modified between the two configuration versions. Figure 2-8 illustrates this change set concept.

Definition of Working Context. SCM systems based on the Change Set model normally rely on the Checkout/Checkin model in defining a working context. A change set is

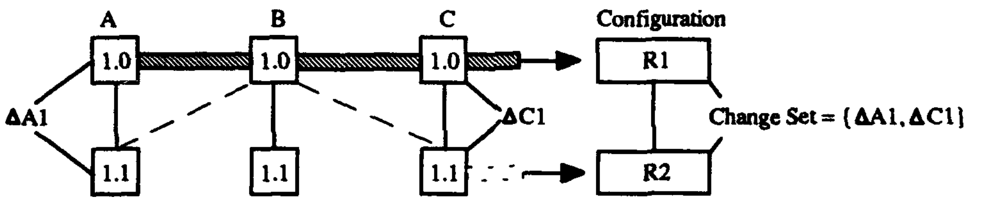

Figure 2-8. Component and Configuration Change Sets (Feiler, 1991:38)

created as part of the working context, then modifications preserved through component check-in are logged as part of the change set (Feiler, 1991:42).

Maintenance of Version History. The first three models maintained the version history of each configuration and component by adding each new version to those already stored in the repository. The resultant version history was reflected in the version graph. SCM systems based on the Change Set model maintain and track the change sets applied against the baseline configuration and its constituent components (Feiler, 1991:40). Therefore, each configuration version can be derived from the baseline configuration and the applicable change set(s), and each component version can be derived from the baseline component and the applicable change set(s).

Control of Concurrent Changes. The Change Set model does not provide any locking mechanisms to control concurrency of changes. Instead, SCM systems based on this model must also support, and thus rely upon, the Checkout/Checkin model for concurrency control (Feiler, 1991:43).

Management and Propagation of Change. This model provides management of logical changes in that each change set serves as a record that persists after the activity creating the change has been completed (Feiler, 1991:37). Changes are propagated to other configurations by including the respective change set (Feiler, 1991:37).

Model Summary. Typically, an SCM system will focus on one of the four identified models as its primary model, possibly complementing it with a second model (Feiler, 1991:45). Regardless of which model(s) an SCM system supports, it must be capable of being integrated with other CASE tools and environment frameworks employed for a given project (Feiler, 1991:46). The tools must be able to pass data back and forth and also be able to communicate with each other if developers are to use the various tools and frameworks effectively. Because no current SCM tool is capable of addressing every SCM challenge, tools and technologies remain popular subjects of research (Whitgift, 1991:155). According to Feiler, ". . . there is a need for a unified CM model that provides a framework for configuration management support. This unified model should be a multi-paradigm model that supports several CM concepts cooperating in harmony" (Feiler, 1991:47).

Summary Of Review

Although SCM has finally gained the recognition it deserves and is considered in the same light of importance as hardware CM, it will continue to be a topic for debate among software managers who are looking for more concrete and efficient ways to manage and control software systems. There is a definite need for better tools and procedures to perform SCM. However, there is also a need for software managers to better understand their particular SCM requirements and how to effectively implement the available tools to meet these requirements. The degree to which each fundamental SCM element is stressed and implemented depends on the individual software system. As Dean states, "The amount and type of detailed information required for your program is your decision; the means of tracking it will be determined by your program, but the tracking must be done" (Dean, 1979:26). Today, there is a myriad of software management tools from which to choose; their presence in any software development environment is commonplace. It is essential that Air Force managers become intimate with their capabilities, or at the very least, be acquainted with their use. This review has presented a compilation of many involved topics that require understanding before any evaluation of SCM tools can begin. The information uncovered during this research and the methodology discussed in the following chapter made it possible to develop an evaluation mechanism for SCM tools.

III. Methodology

Overview

This chapter addresses the methods and techniques used to answer the research questions identified in Chapter 1. First, we discuss the development of the evaluation mechanism, and specifically, how, and from where, DoD SCM requirements were identified. These are the requirements against which each SCM tool was measured. This section also includes a description of the method used to examine and define the functionality of SCM tools in general. For purposes of this study, these requirements and functionality were arranged in a matrix format which became part of the evaluation mechanism. Second, the population of SCM tools is described and followed by the strategy used in selecting a sample from this population. The process used to gather data on each tool in the sample is discussed; specifically, what type of data was used and how it was collected. Finally, the chapter ends with the methodology for using the evaluation mechanism in the analysis of each tool in the sample.

Development of the Evaluation Mechanism

As stated in Chapter 1, this research effort focused on developing an evaluation mechanism for consistently and systematically analyzing SCM tools which could be used in Air Force software systems development. Specific criteria were needed in order to examine and evaluate each tool, consisting of two sets: first, existing SCM requirements and second, general SCM tool functionality, as discussed in Chapter 2. Each of these criteria sets is discussed further in the following sections. Using this evaluation mechanism, each tool in our sample was assessed to determine what functionality it contained, and whether or not it met each respective requirement.

SCM Requirements Analysis. The SCM requirements are primarily based on DoD-STD-2167A, *Defense System Software Development,* and MIL-STD-973, *Configuration Management.* In addition, industry standards were examined, which included: IEEE 828-90, *Standard for Software Configuration Management*, and IEEE 1042-87, *Guide to Software Configuration Management*. We analyzed both DoD-STD-2167A and MIL-STD-973 to determine all the SCM requirements that currently can be levied on new DoD software development programs. This list evolved into the DoD SCM requirements for the evaluation mechanism. The industry standards were also analyzed to obtain a standard requirements list, which was compared to the DoD requirements list in order to identify any unique differences and to increase our understanding of each requirement. An explanation was developed to clarify and justify each requirement. Justification included a reference to the applicable standard(s) specifying each requirement. These requirements apply to a wide range of software development programs and, as the standards state, are meant to be tailored. Requirements that do not apply or are in excess of the program's needs should be eliminated, or tailored, from the requirements list. In order to make the evaluation mechanism applicable to all software development programs, all requirements identified during the analysis of the standards were included in the evaluation mechanism. Where no weighting system was used, each requirement was considered equally important to the SCM effort. In use, the Air Force manager may assign weights to each requirement, depending upon the needs of the particular software development effort.

SCM Tool Fundamentals Analysis. As discussed in Chapter 2, the majority of SCM tools available today utilize four fundamental models, either singularly or in combination, and to varying degrees. Each model was examined to define its specific fundamentals. Also, prominent families of SCM tools were analyzed to identify additional functions or capabilities. This information became the "fundamental functions" used in the evaluation mechanism for this research. In addition to this list, an explanation was developed to provide a better understanding of each fundamental function. Like the SCM requirements, no evaluation weights were applied to these fundamental functions. Rather, each function was considered equally important. It is left up to the Air Force manager to apply weights defined by the unique needs of their software development effort(s).

Constructing the Evaluation Mechanism. From the results of the analyses of SCM and software development standards, and the fundamental SCM models and tool families, a Requirements-Functionality matrix was developed. The rows of the matrix consist of the DoD SCM requirements identified, while the colums of the matrix are comprised of the fundamental functions. Figure 3-1 shows the general format of the matrix.

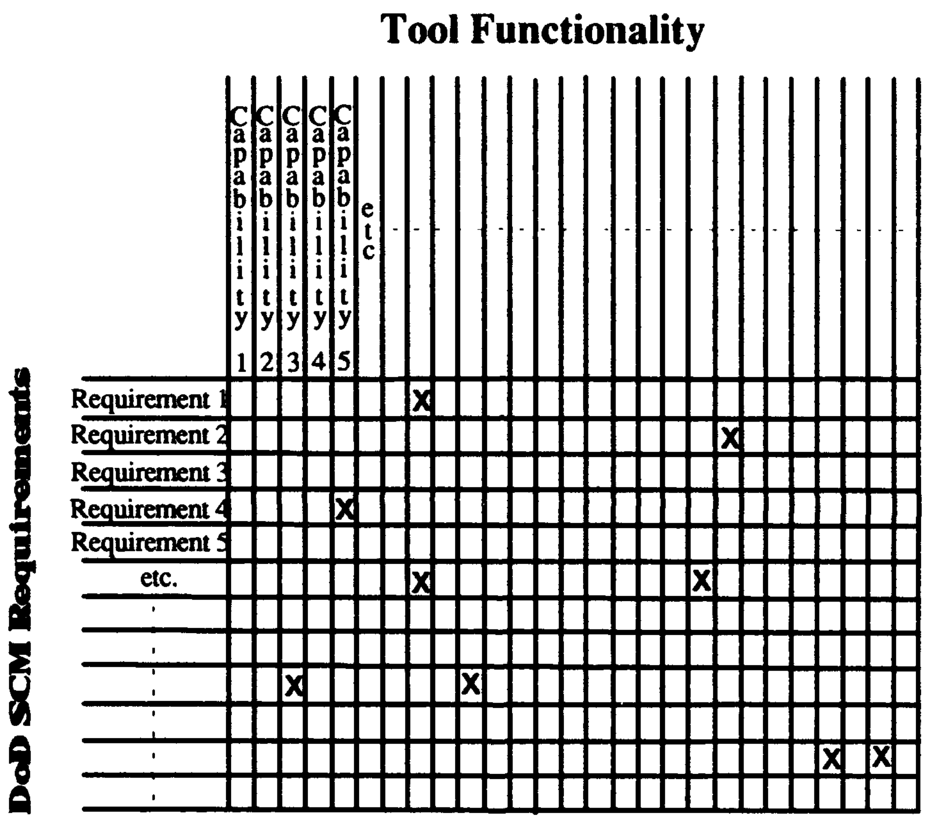

Figure 3-1. Requirements-Functionality Matrix Format

An "X" in a block indicates that the associated tool meets a specific requirement while utilizing a particular functionality. For each tool, the rationale behind that respective determination was discussed to justify how the tool satisfies the particular requirement. This information, along with the Requirements-Functionality matrix and general tool information, comprised the evaluation mechanism.

Tool Evaluation

One of the research objectives was to evaluate a sample of SCM tools using the evaluation mechanism discussed and described in the preceding sections. The scope of this research effort was discussed briefly in Chapter 1. Because of the numerous available SCM tools, the number of tools assessed, using the evaluation mechanism had to be limited. To accomplish this, the population of tools was first identified. From this population a sample of tools was selected . Before beginning the evaluation, an understanding of the capabilities of each tool in the sample was required. This involved gathering, studying, and understanding data specific to each tool in the sample. Once the capabilities and functionality of each tool were understood, we applied the evaluation mechanism and performed a requirements analysis. The individual steps in the tool evaluation process are discussed in the following paragraphs.

Population of Interest. The population of SCM tools that served as the foundation of this research effort was defined by the four criteria identified below.

1. Available Commercial-Off-The-Shelf (COTS). This term implies that the population consisted of tools available to any organization or individual with the funds necessary to purchase the tool. No program or system unique tools will be evaluated.

2. Tools currently used by the Air Force on delivered programs which have a proven track record, whether good or bad, through actual use by Air Force organizations.

3. Tools used by defense contractors in the development of software for the Air Force, ranging from major weapon systems software to management information system software,

4. Tools able to support the Ada programming language. Since the DoD has been mandated to use the Ada Higher Order Language, more and more software will be developed in Ada. Therefore, to be valuable to an Air Force manager, an SCM tool must be able to accommodate any Ada peculiarities.

Sampling Strategy. To ensure a thorough cross-section of the population, various Air Force program offices, defense contractors, previous research, and local vendors were investigated to identify a candidate list of SCM tools from which to choose a sample. The sample was eventually determined by those companies which elected to participate in the research effort by providing data and other support. The tools for which sufficient data could not be located were eliminated from consideration.

Data Gathering. Before gathering data on our tool sample, the specific data required and the strategy needed to obtain this data were defined.

Identification. There were two basic types of data required for each tool in the sample. First, general product descriptions were needed, which provided a general idea of the capabilities of each tool. Second, in order to understand the functionality of each tool, we required more detailed technical data was required, which was in the form of either support and operator manuals, evaluation software provided by the vendors, or dialogue with tool experts.

Strategy. The first type of data required was obtained through product vendors, in the form of sales brochures and popular magazine articles. More in-depth technical information required investigating trade journals, interfacing with the vendor's engineering support, and fully exercising any demonstration disks we received from vendors. For each tool in our sample, the data necessary to understand and identify its specific functions was obtained.

Evaluation and Analysis. Once substantial data had been collected, we conducted an evaluation of each tool. This included examining the data for each specific tool to determine what specific DoD SCM requirements were met using the evaluation mechanism. All tools were evaluated independently of one another. Rationale for each decision made in the matrix was justified or explained during the analysis. Each tool in the sample was assessed to determine its unique features in meeting each of the particular SCM requirements. No tool was chosen as "the best" for two reasons: tools target different portions of the software development life cycle, and it was not our intent in this research effort to specify a "best" tool. The results of our evaluation and analysis are examined in Chapter 4.

Summary

The goal of this research effort was to provide a mechanism by which Air Force managers can better understand SCM requirements, and thus be capable of determining which of the various tools and technologies employed by contractors can best meet these SCM requirements. This required a two stage methodology. First, using known DoD SCM requirements and generally accepted tool functionality, develop a mechanism that can be used to evaluate specific tools. Second, using the mechanism, systematically analyze and evaluate a group of tools. The knowledge gained from this research can help Air Force managers initially specify the necessary SCM requirements for their particular system, and then, competently monitor the contractor's performance to these requirements. The efforts involved in developing the evaluation mechanism, analyzing the selected sample of SCM tools, and identifying the difficulties encountered in the process are all discussed in the next chapter.

IV. Analysis

Overview

This chapter presents an analysis of the data and other information obtained in support of this research effort. Specifically, we will address both the effort involved in developing the software configuration management tool evaluation mechanism, and that involved in actually using the mechanism to assess several commercial-off-the-shelf tools. In several instances, we have attached an appendix of specific data to support our analysis.

Evaluation Mechanism

The brunt of the effort in developing the SCM Tool Evaluation Mechanism lay in constructing the SCM Requirements-Functionality Matrix. This was of particular importance because the matrix depended upon our abilities to accurately compile a list or set of DoD requirements addressing SCM, and then efficiently organize a set of functional areas that realistically represented the potential capabilities which an SCM tool can possess. The remaining portion of this evaluation mechanism development effort involved integrating the matrix with a standardized package format which would provide other useful information regarding the particular tool. In the following subsections, each phase of this effort is described.

SCM Requirements Defined. Both DoD-STD-2167A and MIL-STD-973 were examined to determine the current specific SCM requirements governing the DoD acquisition of software. In addition, IEEE Std 1042-1987 and IEEE Std 828-1990 were examined to identify possible dissimilarities between the SCM requirements prevailing in DoD acquisition, versus those in commercial industry development. As the requirements were compiled, they were grouped according to the four fundamental SCM elements: identification, control, auditing, and status accounting.

Initially, the number of SCM requirements compiled was quite lengthy. We attempted to construct the vertical axis of the SCM Requirements-Functionality Matrix using this initial list of requirements. However, we discovered that, due to the large number of requirements, the matrix extended onto two pages. This was cumbersome and would result in a less effective evaluation mechanism. Therefore, the size of this list was reduced in order that it might fit on one page. DoD-STD-2167A and MIL-STD-973 share many of the same SCM requirements, and, as a result, identical requirements were combined. Furthermore, while many requirements within the same standard were redundant, such requirements were combined into a single requirement, whenever this was feasible. By eliminating requirement redundancy, the list was reduced – but not sufficiently. In order to achieve a list of requirements which would adequately represent SCM responsibilities while efficiently occupying no more than one page, logically related requirements were grouped together under one massaged requirement label. The resulting final list consisted of 33 SCM requirements. Appendix A lists these requirements and provides a cross reference to the applicable standard(s) and paragraph(s) for each requirement. Each group of SCM requirements is now listed and fully described based on DoD-STD-2167A and/or MIL-STD-973.

Configuration Identification Requirements. The list of SCM requirements included 10 requirements addressing configuration identification. These requirements are shown in Table 4-1 and each one is discussed in more detail in the following paragraphs. As mentioned above, a cross reference for each requirement and the specific standard(s) and paragraph(s) is presented in Appendix A.

Document and implement plans for performing configuration identification. The contractor shall document and implement plans establishing identification policies and procedures as outlined in the governing contract. The procedures will address the identification of data files submitted for approval during reviews, audits, or other events or activities established by the contract. Each document, software listing, etc. will be identified by a unique identifier which specifies the version and submittal status. Additionally, the plan and procedures will indicate how changes from previous versions are identified.

Configuration Identification
1. Document and implement plans for performing configuration identification.
2. Select CSCIs.
3. For each CSCI, identify baselines, developmental configuration, and associated documentation.
4. Trace CSCIs to Work Breakdown Structure elements when MIL-STD-881 is invoked.
5. Decompose and partition each CSCI into CSCs and CSUs.
6. Identify and label documentation, software, and software media placed under configuration control.
7. Identify, define, and document interfaces.
8. For each CSCI, allocate and provide traceability for requirements to lower SCIs and documentation.
9. Ensure correlation between each SCI, its documentation, and other associated data.
10. Display information about an identifier upon command.

Table 4-1. Configuration Identification Requirements

Select CSCIs. The contractor shall decompose the system into hardware and software requirements. Similar software requirements shall be logically grouped into computer software configuration items (CSCIs). The type of configuration documentation for each CSCI will be identified.

For each CSCI, identify baselines, developmental configuration, and associated documentation. The contractor shall establish the functional, allocated, and product baselines and developmental configuration for each CSCI. The specific documentation which will be used to establish each configuration baseline is identified. Additionally, the specific documentation that will be controlled internally by the contractor as part of the developmental configuration is identified. Each configuration baseline and developmental configuration will be established at specified points during the system or CSCI life cycle. The establishment of these baselines normally follows government approval of the configuration documentation that describes each baseline.

Trace CSCIs to WBS elements when MIL-STD-881 is invoked. The contractor shall ensure that each CSCI can be traced back to the Work Breakdown Structure (WBS).

Decompose each CSCI into CSCs and CSUs. Each CSCI shall be further decomposed and partitioned into CSCs and CSUs both to facilitate the allocation of requirements down to lower units and to ease both the design and testing later in the life cycle of the CSCI.

Identify and label software, documentation, and software media. The contractor shall obtain or be issued an identifier for each CSCI, CSC, CSU, and documentation item. Each identifier shall consist of a name or number, version, revision, release/release date, type designator, nomenclature, and change status. Each identifier shall be embedded within the applicable software. Software media (i.e., code, documentation, or both) shall be marked with either a label specifying the software it contains, or a cross reference to a listing of the identifiers of that software. Furthermore, the media shall be labeled with the contract number, Contractor And Government Entity (CAGE) code, media or serial number, and lot numbers (if applicable).

Identify, define, and document interfaces. The contractor shall identify and document all interfaces with, or required by, each CSCI. As part of the functional configuration documentation, selected items shall be identified which are to be integrated or interfaced with the CSCI. This may include software (or hardware for tightly coupled systems) developed separately, commercial-off-the-shelf, or already in existence. Interface requirements shall be documented in an Interface Requirements Specification for each CSCI.

For each CSCI, allocate and provide traceability for requirement to lower SCIs and documentation. The contractor shall allocate requirements for each CSCI to its respective CSCs and CSUs to facilitate design, development, and test. The traceability of these requirements shall be documented in each CSCI's Software Requirements Specification (SRS) and

Interface Requirements Specification (IRS) to indicate a flow of requirements from the system level specification to each CSCI, and from each CSCI to its CSCs and CSUs. The traceability also applies to the flow of requirement from the SRS and IRS to test cases identified in the Software Test Description (STD) and shall be documented in the STD.

Ensure correlation between each SCI, its documentation, and other data. The contractor shall ensure, through identification and marking, that each SCI, its documentation, and any other pertinent data correlate with one another. Identifiers shall relate software to its associated design and configuration documentation and shall be documented in a Version Description Document.

Display information about an identifier upon command. The contractor shall provide the capability to display information about a particular identifier. Information might include name or number, version, revision, release/release date, type designator, nomenclature, change status, and associated documentation.

Configuration Control Requirements. The list of SCM requirements included eight general requirements pertaining to configuration control. These requirements are listed in Table 4-2 below. Once again, Appendix A provides the references to the specific standard(s) and paragraph(s) for each requirement listed.

Configuration Control
1. Document and implement plans and procedures for configuration control.
2. Establish an engineering release system.
3. Document and implement a corrective action process.
4. Apply internal configuration control prior to baselining products.
5. Maintain master copies of, and control changes to, deliverable software and documentation.
6. Prepare a problem/change report for each problem detected.
7. Prepare and classify changes to baselined documentation and software.
8. Provide access to documentation and code under configuration control.

Table 4-2. Configuration Control Requirements

Document and implement plans and procedures for configuration control. The contractor shall document and implement plans and procedures for controlling software and its associated documentation contained in the Software Development Libraries (SDLs) established for the contract. These procedures shall provide for controlling databases and files during reviews and update cycles. Procedures shall be in place to control the software and configuration documentation for each SCI prior to, as well as after, being baselined. The plans and procedures shall address the control of software, documentation, and data during the developmental configuration. These procedures shall regulate proposed changes, deviations, and waivers, document the impact and effectivity of proposed changes, and incorporate only approved changes. The plan shall describe the process required to: identify the need for the change/deviation/waiver, classify the change, prepare the required forms, review and evaluate a proposed change/deviation/waiver, and implement an approved change/deviation/waiver.

Establish an engineering release system. The contractor shall establish an engineering release system to control the issuance and authorize the use of documentation associated with an approved configuration. As part of the engineering release system, a release signature shall be included for each CSCI specification, identifying that the document has been reviewed and has been approved for release. When required by contract, a DD Form 2617, "Engineering Release Record (ERR)", will be used to release configuration documentation to the government for approval. All initial releases of, and approved changes to, already released documentation, software, and other data that establishes a baseline shall be accomplished utilizing an ERR.

Document and implement a corrective action process. The contractor shall document and implement a process with procedures to handle problems encountered in products under internal configuration control. The corrective action process shall ensure that all detected problems are quickly reported, action is taken, resolution is achieved, status is tracked and reported, and historical records are maintained for the duration of the contract. The process shall include problem/change reports, classification of problems by category and priority, defect trend analysis of reported problems, and the evaluation of corrective actions.

Apply internal configuration control prior to baselining products. The contractor shall control software and associated documentation prior to being baselined by the government. After each CSU, CSC, and CSCI are successfully tested and evaluated, the contractor shall place the Software Design Document (SDD) and source code listing into the appropriate Developmental Configuration. The SDD for each CSCI shall initially be placed into the Development Configuration after preliminary design. The SDD and source listings are updated as a result of detailed design, CSU testing, CSC integration and testing, and CSCI integration and testing.

Maintain master copies of, and control changes to, deliverable software and documentation. The contractor shall control all software source code and documentation scheduled to be delivered to the government as part of the contract. Master copies of the software code and documentation originals shall be kept current, and the preparation and dissemination of changes shall be controlled. Changes to deliverables will occur only as a result of an approved Class I or Class II change and will utilize an Engineering Release Record. Documentation may include the Software Development Plan, System/Segment Design Document, Software Requirements Specification, Interface Requirements Specification, Software Test Plan, Software Design Document, Interface Design Document, and Software Test Description.

Prepare a problem/change report for each problem detected. For each problem detected in software or documentation that is under internal configuration control, the contractor shall prepare a problem/change report. The report shall fully describe the problem, corrective action required, and the actions taken to resolve the problem.

Prepare and classify changes to baselined documentation and software. The contractor shall prepare Engineering Change Proposals (ECPs) for necessary changes to baselined software and Specification Change Notices (SCNs) for necessary configuration documentation changes. The contractor shall classify ECPs as Class I (preliminary or formal) or Class II in accordance with the criteria established in MIL-STD-973. Unless otherwise contracted, ECPs and SCNs shall be prepared on a DD Form 1692, "Engineering Change Proposal", and DD Form 1696, "Specification Change Notice". An Advance Change Study Notice (ACSN) shall be used prior to the preparation of a formal routine ECP to summarize a change or identify a topic for a change proposal. Also included in this requirement is the preparation of deviations and waivers, which shall be classified as critical, major, or minor in accordance with MIL-STD-973, and shall be requested using a DD Form 1694, "Request for Deviation/Waiver", or other form contractually agreed upon.

Provide access to documentation and code under control. The contractor shall provide the government with access to software and documentation that is under the contractor's internal configuration control.

Configuration Auditing Requirements. Most of the requirements extracted from the standards which addressed configuration auditing pertained to details involving how the government should audit and review contractor SCM efforts, rather than how the contractor should support configuration auditing. Therefore, the list of SCM requirements included only three general requirements addressing configuration auditing. These requirements are listed in Table 4-3 and are discussed in detail following the table. Once again, Appendix A contains references to the specific standard(s) and paragraph(s) for these requirements.

Configuration Auditing
1. Conduct or support formal reviews and audits.
2. Participate in the resolution of discrepancies identified during reviews and audits.
3. Record and publish meeting minutes.

Table 4-3. Configuration Auditing Requirements

Provide information in support of formal reviews and audits. The contractor shall participate in, or chair, formal reviews and audits. These include the System Requirements Review (SRR), System Design Review (SDR), System Specification Review (SSR), Preliminary Design Review (PDR), Critical Design Review (CDR), Test Readiness Review (TRR), Functional Configuration Audit (FCA), and Physical Configuration Audit (PCA). These reviews and audits may be conducted for each CSCI individually, or concurrently for multiple CSCIs. The contractor is responsible for designating a co-chairperson and providing the necessary resources and material required to perform reviews and audits. Also included in this requirement is the contractor's responsibility to establish the time, location, and agenda for each review and audit in accordance with the contract. Required information for each CSCI includes:

- Identification of items to be reviewed or audited.
- Copies of specifications, design documents, software listings, test plans, and procedures and other documents which describe the contents or use of the CSCI.
- Listing of all deviations/waivers.
- Listing of approved and outstanding changes.
- Test data, results, and reports.
- Matrix that identifies requirements of sections three and four of the specifications; includes a cross reference to the test plan, test procedures, and test reports, results of demonstrations, inspections, and analyses of requirements; and identifies all deficiencies.
- Delivery media for software, documentation, or both.
- Internal evaluation and inspection results

Participate in the resolution of discrepancies identified during reviews and audits. The contractor shall record and track discrepancies identified during the conduct of reviews and audits. The contractor shall accomplish residual tasks associated with those discrepancies for which they were identified as being responsible, or assist in determining if ECPs are required.

Record and publish meeting minutes. This requirement is self explanatory; hence no further discussion is provided.

Configuration Status Accounting Requirements. Finally, the list of SCM requirements included twelve requirements addressing configuration status accounting. These requirements are presented in Table 4-4 below. Each requirement listed is further discussed in the following paragraphs.

Configuration Status Accounting
1. Document and implement plans and procedures for performing configuration status accounting.
2. Establish and maintain software development files (SDFs).
3. Establish software and documentation libraries.
4. Provide and control access to development histories.
5. Prepare and maintain management records, status reports, and product evaluation records.
6. Analyze configuration status accounting.
7. Record the current, approved software, documentation, and identifiers.
8. Record and report the status of request for engineering changes, deviations, and waivers.
9. Record and report implementation status of authorized changes.
10. Record and report the location of each CSCI version in the field.
11. Ensure information about new releases is incorporated into the configuration status accounting system.
12. Record and report the results of configuration audits.

Table 4-4. Configuration Status Accounting Requirements

Document and implement plans and procedures for performing configuration status accounting. The contractor shall plan for, and establish procedures for, recording, storing, and reporting data concerning products that comprise the Developmental Configuration and configuration baselines.

Establish and maintain software development files (SDFs). The contractor shall establish a repository containing products and data associated with each SCI developed. Depending on the requirements of the contract, the contractor shall establish SDFs for each CSCI, each CSC or logical group of CSCs, and each CSU or logical group of CSUs. The SDFs will be maintained for the duration of the contract and will contain information such as design considerations, design constraints, design documentation, data, evaluation results, schedules, status, and test information.

Establish software and documentation libraries. The contractor shall establish repositories in which to store software and documentation developed as part of the contracted effort.

Provide and control access to development histories. The contractor shall provide access to development histories depending on applicable distribution codes, security requirements, and Contract Data Requirements List distribution. If digital data is required by contract, query capabilities shall be provided along with procedures defining the control of databases and files during review.

Prepare and maintain management records, status reports, and software evaluation records. The contractor shall generate management records and status reports on all products that comprise the Developmental Configuration and the Allocated and Product baselines. Records shall also be prepared and maintained for every software product evaluation performed.

Analyze status accounting data. The contractor shall review and analyze configuration status accounting data to detect problem trends in reported problems, shall verify problem resolution, and shall ensure that no additional problems have been introduced as a result of the fix.

Record the current and approved software, documentation, and identifiers. The contractor shall establish and maintain records identifying the current approved software, configuration documentation, and identification number associated with each SCI. These records shall also include historical information for items associated with each SCI, such as past revisions to specifications or software versions. For each item just identified, the current status shall be maintained (e.g., working, submitted, approved, released). The contractor shall also record active contracts (subcontractors, vendors, etc.) affecting the program.

Record and report the status of request for engineering changes, deviations, and waivers. The contractor shall record, store, and report the status of all proposed engineering changes and all critical and major requests for deviations and waivers which affect the configuration. The status of engineering changes will be recorded and reported, starting from initial submittal to the government and ending with final approval/disapproval and contractual implementation. These records shall also contain historical data and information. General information describing each proposed change shall be maintained in the records, and specific activities and events associated with the processing of each change shall be tracked.

Record and report implementation status of authorized changes. For each CSCI, the contractor shall maintain records that contain historical information that documents all changes to an approved configuration and configuration documentation. Records shall provide traceability of all changes from the original baselined documentation. The status of all authorized changes shall be reported. The implementation actions of approved changes shall be tracked and recorded. These actions include responsible activities and required tasks to accomplish each change, as well as scheduled dates for completion. Tasks may include software revision, review, and official release. The status of all retrofit changes to existing products that utilize one or more changed CSCIs shall be reported.

Record and report the location of each CSCI version in the field. The contractor shall maintain records of all software configurations released. This information includes the identification of each CSCI configuration, where the CSCI is installed, the VDD number, and the effectivity and installation status of configuration changes.

Ensure information about new releases in incorporated into the status accounting system. The contractor shall ensure that all information about new releases of software and its associated documentation is recorded in the configuration status accounting system.

Record and report the results of configuration audits. The contractor shall record the results/findings, all discrepancies, residual tasks identified, and the scheduled and actual accomplishment dates for each CSCI audited. General information about each action item shall be maintained. The status and suspense dates of actions associated with closing the action item will be tracked. A historical record of configuration audit information will be maintained.

Based on the review of DoD-STD-2167A and MIL-STD-973, it is the researchers' position that the 33 requirements discussed in this section constitute those fundamental SCM requirements currently required by the DoD when developing software. This list of requirements was used in forming the rows of the SCM Requirements-Functionality Matrix. In order to complete the evaluation mechanism, general SCM tool functions had to be defined. The results of this effort are discussed in the next subsection.

SCM Tool Functionality Defined. The review of SCM theory brought forth many of the activities which must be accomplished to enforce the discipline of software configuration management. These activities center around the four fundamental SCM elements of identification, control, auditing, and status accounting. At the same time, our review of the models used in developing current SCM tools suggested the types of activities which SCM tools could be designed to accomplish. Based on these two areas of our review, a list was compiled of functional capabilities which SCM tools may possess, and upon which they can rely, in meeting SCM-related requirements. It is the researchers' opinion that the functional capabilities of SCM tools can be grouped into the following seven areas of functionality:

1. Database Management
2. Configuration Build
3. Decomposition Control
4. Work Area Control
5. Change Control

6. Baseline Management

7. Customization

Relying on concepts presented in Chapter 2, the seven areas of functionality listed are defined and discussed in the following paragraphs.

Database Management. Repository or library control and querying are key elements of database management. SCM systems that possess this type of functionality provide the systematic and organized storing of the product's software components, whether source code, object code, or documentation. Such tools usually provide this capability by enabling the tool user to create and maintain some kind of database system. Whereas older database technology revolves around a system of multiple directories and sub directories, newer technology calls for an object-oriented relationship approach. SCM tools based on a directory/sub directory database system often utilize the scheme of promoting SCIs from one variant branch (i.e., sub-directory) to another or from one library (i.e., directory) to another to emulate the maturation of the SCIs through their development life cycle. When based upon a relational database structure, the tool allows the user to implement the software items as objects, while creating and managing relationships between these objects. Relational database technology permits the tools to manage the hierarchical relationships between software items and also to manage component versions and life cycle status. Such a relational structure allows for status information to be obtained by querying the database. Regardless of which technology is utilized, the database system can be composed of either a central repository storing all software items, or multiple libraries, each storing a configuration or variant form of the product's software items, or a combination of both. On the other hand, instead of providing for the establishment of a stand-alone database system, a tool can be designed to openly interface with, and depend upon, a database system which pre-exists in the software engineering environment.

Configuration Build. SCM systems that provide configuration build functionality facilitate the user's efforts to build the product configurations which are required for either test or release purposes. In general, tools with this functionality enable the user to input selection rules detailing which component versions are to be included in a given configuration. Based on this input, such tools would automatically collect the applicable components, generate the corresponding derived elements (i.e., object code and linked executables), and assemble these elements into the required configuration.

Decomposition Control. SCM systems that address decomposition control facilitate the tool user's effort to break down the overall system software into separately manageable and less complex parts. Tools possessing this functionality will usually expedite the user's effort to decompose the overall software system into various hierarchical levels of SCIs. Such tools will sometimes automate SCI identification and labeling. Additionally, tools with this functionality may also automatically map each SCI to a Work Breakdown Structure (WBS) element.

Work Area Control. Concepts such as access control, working context, and concurrent/parallel development are key elements of work area control. This area of functionality deals with how an SCM system controls the tool user's development efforts in the work area. The degree of work area control is predicated upon how an SCM system establishes and then controls the working context, or development/working environment. SCM systems routinely force modifications and development work to be performed outside the protected database of SCIs. Developers are forced to copy either individual items or entire system (or subsystem) configurations to personal file directories (i.e., work areas) where they can perform development work and/or implement changes. Access to the SCIs within the repository is strictly controlled. Utilizing various access control rules, which can usually be defined by the tool user, SCM systems can automate the process of determining who can access the system software. Furthermore, SCM systems can limit the number of developers who can concurrently access the same software item(s) and thereby control modifications of the software item(s). Some SCM systems can control and maintain the development history within each team member's work area. In this manner, configuration control is maintained locally, even if only temporarily, outside the protected database of software items. SCM systems that allow multiple users to access software items concurrently provide a means by which to resolve conflicting changes to a software item when they are saved back into the database as a new version (or variant). In such cases, an SCM system can rely on user defined rules to control the merging of conflicting changes so that the result is one new and identifiable version (or variant) of an item.

Change Control. Closely coupled with work area control, change control involves the management of the overall change process. This area of functionality includes key concepts such as version and variant control, the propagation of changes, and the automation of record keeping activities. Version and variant control involves preserving each modified software item as a new version or variant (whichever is applicable) of the item originally copied out of the protected database. SCM systems that provide for version control ensure that when product components are modified, the resulting components and/or the changes themselves (i.e., deltas) are stored as new versions. SCM systems that provide for variant control ensure that product components and/or configurations can be modified and evolved along concurrent development branches. Thorough version and variant control ensures that both the genealogical and version histories of each software entity associated with a product is traceable. As part of change control, some SCM systems will provide a mechanism permitting changes made to one software component or configuration to be propagated to other applicable component or configuration variants supported by the system. This serves to automate and simplify the change effort by eliminating the need to manually implement redundant changes in other affected components. Additionally, some SCM systems will allow logically related changes to be managed, thereby permitting such changes to be identified and easily accessed at a later point in time. For example, if management desired to research which software component versions resulted from a change request, they could simply query the appropriate database of software items by referencing a given logical change (i.e., change request). Finally, SCM systems providing change control functionality usually automate most record-keeping activities involved with performing changes. Change tracking documents (e.g., ECPs, SCNs, deviations, waivers, etc.) can be implemented as objects which can, in turn, be tracked and managed by the SCM tool. Such objects, like their SCI implementation counterparts, can be propagated through the change process life cycle. Often, an SCM tool with this facility provides or utilizes a pre-existing network mail system to prompt applicable user parties (e.g., reviewers, developers, quality assurance, etc.) for input during the phases of a change tracking document's life cycle.

Baseline Management. SCM systems that provide for baseline management enable the tool user to quickly retrieve the constituent SCIs of an officially approved baseline, or the constituent SCIs of an internally defined baseline, or the SCIs which satisfy one or more user-defined selection criteria. Examples of official baseline configurations include functional, allocated, and product baselines. The design baseline, developmental configuration, and test configuration are all examples of internal baseline configurations. Regardless of the type of baseline or configuration, baseline management permits the user either to manage the respective software items together within the protected database or, given component/configuration selection rules, to retrieve the appropriate items comprising a user-defined configuration or baseline. Furthermore, SCM tools with this facility enable the user to copy groupings of items from the controlled repository to a release directory in the work area.

Customization. This area of functionality serves to magnify the scope of capability of the other six functionalities. Tools that possess customization functionality provide the user with the flexibility to fine-tune the tool's facilities to better meet the user's needs and interface more efficiently with the existing software engineering environment. When a tool is highly customizable, the user can then utilize the tool powerfully and extensively to automate SCM responsibilities by defining component design life cycles, change document/record formats, change process life cycles, etc.

SCM Tool Evaluation Mechanism. Based on our compilation of SCM requirements and our definition of a basic set of SCM tool functionalities, the SCM Requirements-Functionality Matrix was developed, and is presented in Figure 4-1,. The final SCM Tool Evaluation mechanism, presented in Figure 4-2, ties the SCM Requirements-Functionality Matrix together with a compilation of background information concerning the tool. In particular, the evaluation mechanism addresses information concerning the product, the vendor, interfacing platforms/operating systems, supported programming languages, and support rationale which substantiates the corresponding matrix evaluation. The next section highlights the results of our efforts in exercising the SCM Tool Evaluation Mechanism while assessing the commercial-off-the-shelf tools sampled.

Tool Name: ____________________ Vendor: ____________________	Database Management	Configuration Build	Decomposition/development Control	Work Area Control	Change Control	Baseline Management	Customization
I. CONFIGURATION IDENTIFICATION							
1. Document and implement plans for performing configuration identification.							
2. Select CSCIs.							
3. For each CSCI, identify baselines, developmental configuration, and associated documentation.							
4. Trace CSCIs to WBS elements when MIL-STD-881 is invoked.							
5. Decompose each CSCI into CSCs and CSUs.							
6. Identify and document the version of each SCI corresponding to the documentation.							
7. Identify, define, and document interfaces.							
8. For each CSCI, allocate and provide traceability for requirements to lower SCIs and documentation.							
9. Ensure correlation between each SCI, its documentation, and other associated data.							
10. Display information about an identifier upon command.							
II. CONFIGURATION CONTROL							
1. Document and implement plans and procedures for configuration control.							
2. Establish an engineering release system.							
3. Document and implement a corrective action process.							
4. Apply internal configuration control prior to baselining products.							
5. Maintain master copies of, and control changes to, deliverable software and documentation.							
6. Prepare a problem/change report for each problem detected.							
7. Prepare and classify changes to baselined documentation and software.							
8. Provide access to documentation and code under configuration control.							
III. CONFIGURATION AUDITING							
1. Provide information in support of formal reviews and audits.							
2. Participate in the resolution of discrepancies identified during reviews and audits.							
3. Record and publish meeting minutes.							
IV. CONFIGURATION STATUS ACCOUNTING							
1. Document and implement plans and procedures for performing configuration status accounting.							
2. Establish and maintain software development files (SDFs).							
3. Establish software and documentation libraries.							
4. Provide and control access to development histories.							
5. Prepare and maintain management records, status reports, and software evaluations records.							
6. Analyze configuration status accounting data.							
7. Record the current, approved software, documentation, and identifiers.							
8. Record and report the status of request for engineering changes, deviations, and waivers.							
9. Record and report implementation status of authorized changes.							
10. Record and report the location of each CSCI version in the field.							
11. Ensure information about new releases is incorporated into the status accounting system.							
12. Record and report the results of configuration audits							

Figure 4-1. SCM Requirements-Functionality Matrix

Software Configuration Management (SCM) Tool Evaluation

Tool Name:
- Version Number:
- Release Date:
- Frequency of Updates:
- Date of First Release:
- Number Sold:

Vendor:
- In Business Since:
- Address:

- Point of Contact:
  - Phone Number:
  - FAX Number:
  - Email Address:

Platforms/Operating Systems:

Programming Languages Supported:

Description: *(basic vendor description of tool)*

SCM Requirements-Functionality Evaluation Matrix: *(attached)*

Substantiation of SCM Requirements met by Tool Functionality: *(Each requirement satisfied by one or more areas of functionality will be substantiated based on technical data from the vendor)*

I. Identification

II. Control

III. Auditing

IV. Status Accounting

Comments: *(any additional information concerning tool capabilities and/or limitations not highlighted in matrix or substantiation)*

Figure 4-2. SCM Tool Evaluation Mechanism

Using the Evaluation Mechanism

Once constructed, the evaluation mechanism was used in an actual assessment of SCM tools in the sample. As stated in Chapter 3, the intent of using the evaluation mechanism developed was not only to evaluate the tools, but also, and perhaps more paramount, to evaluate our evaluation mechanism.

Tool Sample Defined. This research effort focused only on SCM tools that were commercially available. Candidate tools included those which either have been involved in supporting an Air Force software development organization, used on an Air Force software development effort, or identified by Air Force organizations as potential candidates to be utilized in either of the two previous situations. The Software Technology Support Center (STSC) at Hill Air Force Base, Utah, furnished an extensive list of SCM tool vendors, including points of contact. Unfortunately, many of these were either outdated or had no corresponding telephone number. However, nine SCM tool vendors were successfully contacted to assist us in our thesis effort. Background information concerning these vendors, including the product name, point of contact, address, and telephone number, is shown in Appendix B. From the products of these nine vendors, three were selected to perform an initial validation of the evaluation mechanism. The tools selected were Aide-De-Camp (ADC) from Software Maintenance and Development System, Inc., Product Configuration Management System (PCMS) from SQL Software Limited, and CCC/Manager from Softool Corporation. This selection was based upon the detail and completeness of data obtained, vendor support, and the availability of tool documentation or demonstration software. Due to the complexity of the tools and the limited time available for this research effort, the sample was reduced by eliminating CCC/Manager from the list of tools identified above. Both ADC and PCMS are currently being used on Air Force software development programs. The results of using the evaluation mechanism with each tool are discussed in the following section.

Tool Evaluation. The detailed assessments for both ADC and PCMS , each embodied by the completion of the evaluation mechanism for the particular tool, are not presented in this section.

Instead, this material is included in Appendices C and D, respectively. Highlights of both evaluation efforts and results are addressed here.

Aide-De-Camp (ADC). Aide-De-Camp from Software Maintenance and Development Systems, Inc. is an object-oriented tool that utilizes a relational database. Entities such as source code, object code, documentation, design diagrams, test cases, and reports are identified, stored, and tracked as objects and also as files. Objects are associated (linked) with other objects. The files are stored in the ADC relational database and the objects are grouped or categorized logically. For example, source code, documentation, and other information describing or implementing a specific function, such as an improved radar-jamming module for an aircraft, are grouped logically in one directory. This group of logically related entities is defined as a change set (*cset*). ADC recognizes a version as a base (initial) version and a collection of *csets* that describe and document specific changes made to that base version. Versions can be created by adding or subtracting specific *csets* to a base version. ADC supports simultaneous access to files, multiple projects, merging development paths, conflict detection, automated configuration builds, and database security.

Aide-De-Camp was evaluated by using the evaluation mechanism described earlier. The results of this evaluation are presented in Appendix C. Aide-De-Camp was found to meet all but two of the SCM requirements identified and extracted from DoD-STD-2167A and MIL-STD-973. The two SCM requirements from the Requirements-Functionality Matrix that were not met were requirement I-2, "Select CSCIs", and requirement IV-6, "Analyze configuration status accounting data." The particular functionality or functionalities possessed by Aide-De-Camp which enabled it to meet each specific SCM requirement are also identified. This provided insight into which functionalities Aide-De-Camp used to meet each SCM requirement. In addition to the completed matrix, Appendix C provides further substantiation for each matrix result. The evaluation of Aide-De-Camp and the substantiation of the Requirements-Functionality matrix results were based on the information obtained from the *ADC/CM Model 209 User's Guide*, *ADC Command Reference*

Guide, ADC Tutorial for UNIX Systems, ADC User's Guide, X-ADC Administrator's Guide, and *X-ADC User's Guide.* ADC/CM Model 209 is a utility program and X-ADC is the tool's graphical user interface. These applications, and the documents listed, are provided with the Aide-De-Camp tool.

The results from the Requirements-Functionality matrix indicate that Aide-De-Camp relies heavily on the Database Management functionality to meet, at least partially, 27 of the 33 SCM requirements listed on the matrix. The next most used functionality, Customization, was identified for only 12 SCM requirements. Even so, Customization was identified two to three times more than each of the remaining functionalities. From another perspective, a large concentration of functionality is used to meet the requirements identified in configuration identification and configuration control. This indicates that Aide-De-Camp focuses more project identification and control wherever these items are critical for success in a team development environment. Since status accounting deals primarily with record keeping, reporting, and other functions pertaining to database functions, it is not surprising to see that most status accounting requirements were met using the Database Management functionality, rather than any of the other six functionalities.

Product Configuration Management System (PCMS). Based on high-level technical documentation (*PCMS Overview, Edition 2.1*) received from SQL Software Ltd., and several telephone conversations with their senior technical representative, an evaluation of PCMS was performed using the evaluation mechanism. In general, PCMS is capable of supporting a product's entire life cycle, including system decomposition, design and development, test and evaluation, production, and maintenance cycles. Additionally, PCMS supports the configuration of not only software items (code and documentation) but also hardware items. This is a significant capability in that most systems are comprised of both software and hardware requiring that CM efforts for a system address both software and hardware portions simultaneously. Therefore, CM responsibilities for a system cannot be divided out easily into separate efforts.

PCMS is based upon relational database management technology, which permits the entire product to be modeled as objects within the database. The objects and their inter-relationships are predicated on the attributes assigned to each object. The evolution and metamorphosis of the product configuration(s) is therefore managed through the modification of object attributes and the resultant creation of new objects and new inter-relationships. PCMS's relational database facilitates the decomposition of the overall system functionality into *design-parts* which, in turn, are implemented as *product-items* (i.e. SCIs, hardware tracking forms). Furthermore, any type of data (i.e. change tracking forms, meeting minutes, etc.) that can exist on magnetic media can be implemented as a *product-item*, and controlled and managed as an object within PCMS's database system. Layered upon its relational database system, PCMS provides product design and development, life cycle management, change management, release management, team role management, and configuration build facilities. In turn, these PCMS facilities can be integrated with the user's software engineering environment and interfaced to other design, manufacturing, and management tools.

This assessment indicated that PCMS relied – at one point or another – on each of the seven elements of functionality to meet 32 of the 33 compiled SCM requirements. As was the case wit[illegible]C, PCMS failed to meet requirements I-2, "Select CSCIs". It should be noted that, despite PCMS's inability to fully satisfy this SCM requirement, it did provide the facilities to assist the user's effort in meeting this requirement. PCMS's database management functionality, based on state-of-the-art relational database management technology, was the tool's strongest contributing area of functionality, helping PCMS to meet 17 requirements. Another heavily contributing area was PCMS's customization functionality, which helped to meet 15 requirements. The evaluation of PCMS is detailed in Appendix D.

Unfortunately, due to the lack of detailed documentation (i.e. user's manuals), the evaluation of PCMS was limited to a top-level view of how the tool's functionality, rather than implementation capabilities, addressed each SCM requirement.

Tool Assessment Findings. The results of the two independent evaluations were compiled and compared. This comparison is presented in Table 4-5. The evaluation results revealed similar trends between the two tools in the number of requirements met using particular functionalities.

Functionality	Aide-De-Camp	PCMS
Database management	27	17
Configuration build	5	1
Decomposition control	4	6
Work area control	4	2
Change control	6	6
Baseline management	5	4
Customization	12	15

Table 4-5. Requirements Met by Area of Functionality

The numbers in the table indicate that both tools heavily utilize the database management functionality, especially Aide-De-Camp. These results are not surprising since both tools are based on relational database technology. As such, database management is a key area. Also, early SCM tools were primarily databases for storing and retrieving information, namely source code. The results show that the two modern tools of this evaluation are built upon established and proven concepts of their predecessors.

Customization is also widely used by both tools. This is mainly due to the customization functionality overlapping most of the other functional areas. Both tools provide the user with the capability to develop user-defined procedures or commands to customize change control, database management, etc., for their particular organization.

The remaining five functional areas are used much less than database management and customization. Thus it may appear that only database management and customization are crucial in meeting DoD SCM requirements, while the others contribute very little. However, these five functional areas may actually highlight the true usefulness of a tool in today's team development environment where control of the multiple configurations is essential. Table 4-6 below shows the average number of functional areas used to meet each SCM element (i.e., configuration identification, configuration control, etc.). This information provides the manager with a overview of which requirement(s) a particular tool places most emphasis. The larger the number in Table 4-6, the more built in capabilities are provided by the tool and the more flexibility and utilities are available to the user. The average number is calculated by totaling the number of occurrences of all functional areas (the number of marked blocks) in a given SCM element and then dividing by the number of requirements within that SCM element.

SCM Element	Aide-De-Camp	PCMS
Configuration Identification	1.9	1.6
Configuration Control	2.25	1.5
Configuration Auditing	1.66	1.66
Configuration Status Accounting	1.75	1.58

Table 4-6. Average Number of Functional Areas Meeting Requirements

The table indicates that Aide-De-Camp emphasizes configuration control over the other three SCM elements. The number 2.25 indicates the functional density, or average number of functional areas used by Aide-De-Camp to meet each requirement under the configuration control heading. This shows that Aide-De-Camp utilizes a wider variety of functional areas to support configuration control of software products. PCMS appears to address each SCM element equally, as evidenced by the rather consistent functional densities between the elements. This indicates that PCMS is a general tool which adequately addresses all SCM elements but does not concentrate on one in particular. If a project requires considerably more effort in configuration status accounting than in any other SCM element, then the manager should focus on tools that have a higher functional density in the status accounting area.

Summary

Based on the research performed and discussed previously in Chapter 2, the intent was to develop a mechanism to be used when assessing SCM tools. The initial compilation of DoD SCM requirements proved to be too cumbersome, and actually would have inhibited the effective use of the evaluation mechanism. The SCM requirements list was condensed by combining similar and redundant requirements into a single, more general requirement. There was concern that, by grouping requirements, the evaluation mechanism would be less effective since a single requirement on the Requirements-Functionality matrix may actually consist of multiple hidden requirements. Initially, it was believed that the detail obtained by presenting each requirement individually, and as written verbatim from the standards, would ease the difficulty in determining if a requirement was met. This proved not to be the case. The utility of a shorter and more manageable matrix outweighed the detail sacrificed by grouping requirements.

General tool functionalities were determined based on the research of tool theory. Settling on an agreed set of functionalities was as challenging as defining the SCM requirements. A tool's functional capabilities tend to overlap and be interdependent making it difficult to specify stand-alone functional areas. When functional areas were finally defined, they, and the SCM requirements, were married into a Requirements-Functionality matrix that served as the heart of the evaluation mechanism.

Assessments were performed on two SCM tools in order to validate, or fine-tune, the evaluation mechanism. Not all of the facilities of either tool fit precisely into one of the areas of functionality. This resulted in an iterative process of assessing each tool, modifying the evaluation mechanism, and then reassessing each tool using the newly modified evaluation mechanism. During this process, it quickly became apparent that to adequately assess each tool, a working knowledge and possibly even access to a technical expert were necessary. Once completed, the evaluation mechanism provided information that can be useful to a manager interested in a set of tools for a specific software development program. As discussed at the end of the previous section, the evaluation mechanism identifies functional areas and SCM elements where tools are more focused. This can be beneficial to a program whose requirements have been tailored and specific needs are known.

V. Conclusions and Recommendations

Overview

The intent of this research effort was to develop an evaluation mechanism that would permit Air Force managers to make more enlightened decisions when selecting one of the numerous commercially available SCM tools to support his or her program. This chapter presents the overall findings of our research effort. These findings address the analysis results of both the evaluation mechanism and tool evaluation portions of our research effort, the limitations of our research design, and recommendations for further study regarding the evaluation mechanism and tool evaluation.

Analysis Results

Compiling a list of SCM requirements based on published standards and guidelines and compiling a set of fundamental tool functionality based on SCM theory and state of the art tool technologies enabled the construction of the SCM Requirements-Functionality Matrix. Combining this matrix with a generated set of additional background information permitted the development of an effective SCM Tool Evaluation Mechanism. This evaluation mechanism was then used to assess two commercially available SCM tools, Aide-De-Camp and the Product Configuration Management System. The following subsections present the findings regarding the development and utilization of the evaluation mechanism.

Evaluation Mechanism. The evaluation mechanism required numerous iterations before its format finally converged to one which appears to provide a useful and meaningful assessment tool. The final evaluation mechanism, as presented in Chapter 4, is capable of providing information about the general characteristics and capabilities of an assessed tool. A tool's functional areas are mated to the general SCM requirements of a given software development effort. As stated in the standards, the requirements are intended to be tailored or eliminated if not required for a particular program. By grouping the requirements by SCM elemental area, as done in the evaluation mechanism, minor tailoring of requirements will have a negligible impact on the Requirements-Functionality matrix. Only when substantial areas are tailored or removed is the matrix affected. When this does occur, those inapplicable requirements can simply be eliminated or crossed off the matrix. If the evaluation mechanism is used as intended, the evaluator of a candidate SCM tool will have an in-depth understanding of the tool and what it can offer to an organization or program. As evidenced in Appendices C and D, a completed evaluation can be lengthy and involve great detail. Because of this, even if the evaluation mechanism cannot identify which possible SCM tool is the best choice, the mechanism will force the evaluator to focus on the specific SCM requirements of their program and what specific tool functionalities are needed.

Tool Evaluation. Performing the two tool evaluations using the evaluation mechanism provided not only a sincere appreciation for the commercially available technology addressing SCM, but also constructive feedback concerning the merit of the evaluation mechanism. The two evaluations indicated that the notion of meeting one or more SCM requirements did little or no justice in reflecting the tremendous functional capabilities of these tools. Additionally, the tool evaluations indicated that it was difficult to categorize each of a tool's facilities as part of one fundamental functional area.

As mentioned at the end of Chapter 4, actually performing the tool evaluation and completing the evaluation mechanism proved to be difficult and sometimes tedious. A detailed working knowledge of the tool was required. This can be obtained in one of two ways: either acquire a copy of the tool and use it, or read and digest the information contained in vendor supplied documentation. This effort was forced to rely on the latter of the two. The acquired documentation was, in many instances, confusing. Numerous phone calls were made to the vendor of each tool in order to clarify or elaborate upon specific capabilities which had eluded us in our quest to gain a thorough understanding.

Limitations

It should be noted that the proposed research design for this effort had several limitations, which served to decrease the sample size and, in all probability, introduce errors into the overall assessment. First, inaccessible or proprietary data hindered efforts to collect the necessary data needed to develop a working knowledge of a particular tool and to accurately assess the tool. Second, most SCM tools exist as an integrated part of a larger software engineering tool, rather than as a stand-alone tool. Therefore, isolating and extracting SCM-specific information from the overall documentation of the engineering tool proved difficult. Finally, considerable time was required to collect necessary data and to develop a working knowledge of each tool. Therefore, the time constraints of this research effort limited the number of SCM tools that could be assessed and the working knowledge gained for each tool.

Further Study Concerning the Evaluation Mechanism

First, the evaluation mechanism was tuned based on the examination of only two SCM tools. The overall effectiveness of this mechanism over a wide range of SCM tools remains unknown. It would therefore be worthwhile to perform a more in-depth study involving a larger sample of tools, either to validate the evaluation mechanism as is or to determine how it might be changed to improve its effectiveness.

The evaluation mechanism was developed, focusing mainly on DoD SCM requirements, and specifically on how a sample of tools could meet these requirements. This makes good sense. A potential user of any given SCM tool will obviously be interested in whether or not that particular tool will meet their specific SCM requirements. However, in all probability, a user will be equally concerned with the functionality of a tool. For example, how easy is the tool to use in the work area environment and how well does the use of the tool support and/or facilitate software development and modification? These functionality issues do not directly tie into meeting SCM requirements. Therefore, a second area for further study would involve modifying the evaluation mechanism to emphasize tool functionality, or the results and impact thereof, to the same degree as SCM requirements.

The evaluation mechanism developed in this study can only be employed to determine whether or not each SCM requirement is met and if so by which functional area(s). There is no determination as to how well a requirement is or is not met. For example, one tool may be very strong in meeting a given requirement while another tool is very weak in meeting the same requirement. Given this scenario, the current evaluation mechanism would indicate that both tools meet the SCM requirement rather than one tool meeting the requirement decisively over the other. In all likelihood, a user will be quite interested in just how well a particular tool will meet certain SCM requirements, especially those requirements the user emphasizes. Therefore, further study in this area could address incorporating a scheme into the evaluation mechanism based on some established criteria, to indicate the level at which a tool meets a requirement.

Further Study Concerning Tool Evaluations

As stated before, the assessment of each SCM tool was based on a review of vendor supplied documentation such as user's manuals and product literature. Given the proprietary status of the product software and the time limitation governing this research effort, it was not possible to obtain a working copy of either tool and then develop a true working knowledge of each. This is probably not unlike the situation in which Air Force management would be if they were looking at assessing a sample of tools.

Therefore, the tool evaluations were based on the limited understanding of each tool. It would be worthwhile to research these same two tools, as well as other tools, by surveying actual Air Force and other DoD programs which have, either themselves or through an outside contractor/organization, used these tools to enforce SCM requirements. In this manner, valuable input from the knowledgeable user(s) of a tool could be meshed into that tool's evaluation providing a more accurate portrayal and substantiation of that tool's functional capabilities and ability to meet the various SCM requirements.

Summary

This research study was performed, primarily, in an effort to develop an evaluation mechanism to assist Air Force management in selecting a software configuration management tool for their program or project. Only by field-testing the evaluation mechanism in actual program offices can the effectiveness of this effort be verified. There are important points to remember when considering the use of the evaluation mechanism developed as a result of this research.

First, an organization must follow a defined tool assessment process. As a minimum, this process should consist of analyzing the organization's needs, analyzing the environment in which the tool will operate, developing a tool candidate list, and then applying criteria and selecting a tool (Firth et al, 1987:31-33). Therefore, an evaluation mechanism, such as the one developed as part of this research effort, is only a part of a much larger assessment process.

Second, an organization must have a well defined configuration management process in place which is understood by everyone involved. If an organization does not have a mature and enforced process, then attempting to assess the potential impact of a variety of SCM tools is pointless if not impossible (Paulk et al, 1991:4).

Third, using this evaluation mechanism will not guarantee that the final tool chosen will improve the SCM environment of an organization. IEEE Std 1042-1987 asserts that if the members of an organization do not trust or are unwilling to use new SCM tools and methods then the organization's performance will not improve and may, in fact, be hindered (ANSI/IEEE, 1988:33). Therefore, the management of any organization must carefully consider the decision to proceed with the time consuming and expensive process of assessing and procuring an SCM tool.

In conclusion, it is our hope that, as a result of this research effort, not only can Air Force management make worthwhile use of this evaluation mechanism but also that it can gain a more cognizant understanding of the numerous SCM requirements, the importance of these requirements to the successful accomplishment of any software development effort, and the nature and potential of the tool technology commercially available.

Appendix A: SCM Requirements - Standards Cross Reference

The information contained in this appendix provides a listing of each SCM requirement used in the evaluation mechanism. Next to each requirement, the standard (DOD-STD-2167A, MIL-STD-973, or both) from where the requirement was determined is cited and the specific paragraph(s) is identified. For readability, "2167A" refers to DOD-STD-2167A and "973" refers to MIL-STD-973. In both cases, each is followed by the specific paragraph being referenced.

<u>Configuration Identification Requirements</u>	<u>Reference</u>
1. Document and implement plans for performing configuration identification.	2167A, 4.5.1 973, 4.3.2.c.1 973, 4.3.2.c.3
2. Select CSCIs.	973, 4.4 973, 5.3.1.a 973, 5.3.2
3. For each CSCI, identify baselines, developmental configuration, and associated documentation.	973, 4.4 973, 5.3.1.b 973, 5.3.1.e 973, 5.3.1.f 973, 5.3.4 2167A, 4.5.1.a 2167A, 4.5.2.a 2167A, 5.7.5.2
4. Trace CSCIs to Work Breakdown Structure elements when MIL-STD-881 is invoked.	973, 5.2.2
5. Decompose each CSCI into CSCs and CSUs.	2167A, 4.2.5
6. Identify and label documentation, software, and software media placed under configuration control.	2167A, 4.5.1.b 2167A, 4.5.1.c 2167A, 4.5.1.d 2167A, 4.5.1.f 973, 4.4 973, 5.3.1.g 973, 5.3.1.i 973, 5.3.6 973, 5.3.6.1 973, 5.3.6.2 973, 5.3.6.3 973, 5.3.6.5 973, 5.3.6.7.1
7. Identify, define, and document interfaces.	973, 5.3.1.d 973, 5.3.4.1.1 973,5.3.7.1

Configuration Identification Requirements (cont.)	Reference
8. For each CSCI, allocate and provide for requirements to lower SCIs and documentation.	2167A, 4.2.5 2167A, 4.2.6 2167A, 4.3.4
9. Ensure correlation between each SCI, its documentation, and other associated data.	2167A, 4.5.1.e 2167A, 5.7.5.1 973, 5.3.1.h 973, 5.3.6.5
10. Display information about an identifier upon command.	973, 5.3.6.5

Configuration Control Requirements	**Reference**
1. Document and implement plans and procedures for configuration control.	2167A, 4.1.8 2167A, 4.5.2 973, 4.3.2.c 973, 4.3.3 973, 4.5 973, 5.3.3 973, 5.3.3.3 973, 5.4.1 973, 5.4.2.1
2. Establish an engineering release system.	973, 5.3.5 973, 5.3.5.1 973, 5.3.5.2
3. Document and implement a corrective action process.	2167A, 4.1.9 973, 5.3.3
4. Apply internal configuration control prior to baselining products.	973, 4.5
5. Maintain master copies of, and control changes to, deliverable software and documentation.	2167A, 4.5.2.b 2167A, 4.5.2.d 2167A, 5.1.5 2167A, 5.2.5 2167A, 5.3.5.2 2167A, 5.3.5.3 2167A, 5.4.5.2 2167A, 5.4.5.3 2167A, 5.5.5.2 973, 5.3.4.2
6. Prepare a problem/change report for each problem detected.	2167A, 4.1.10 2167A, 4.4.3 2167A, 5.3.5.1 2167A, 5.4.5.1 2167A, 5.5.5.1 2167A, 5.6.5 973, 5.3.3

Configuration Control Requirements (cont.)	Reference
7. Prepare and classify changes to baselined documentation and software.	2167A, 4.5.5 2167A, 5.8.5 973, 5.4.2.1 973, 5.4.2.2.1 973, 5.4.2.3.3 973, 5.4.2.3.3.1.2 973, 5.4.3.3 973, 5.4.4.3 973, 5.4.6
8. Provide access to documentation and code under configuration control.	2167A, 4.5.2.c

Configuration Auditing Requirements	Reference
1. Provide information in support of formal reviews and audits.	2167A, 4.1.2 2167A, 5.1.1.1 2167A, 5.1.1.2 2167A, 5.2.1 2167A, 5.3.1 2167A, 5.4.1 2167A, 5.7.1 2167A, 5.8.1 973, 4.7 973, 5.2.3 973, 5.6.1 973, 5.6.1.1 973, 5.6.1.2 973, 5.6.1.3 973, 5.6.2 973, 5.6.2.1 973, 5.6.2.2 973, 5.6.2.3 973, 5.6.3 973, 5.6.3.1 973, 5.6.3.2 973, 5.6.3.3.f 973, 5.6.3.3.g 973, 5.6.3.3.h
2. Participate in the resolution of discrepancies identified during reviews and audits.	973, 4.7 973, 5.6.2.4.c 973, 5.6.3.4.c
3. Record and publish meeting minutes.	973, 5.6.1.3.e 973, 5.6.2.4.a 973, 5.6.3.4.b

Configuration Status Accounting Requirements	**Reference**
1. Document and implement plans and procedures for performing configuration status accounting.	2167A, 4.5.3 973, 4.2.9 973, 4.3.2.c
2. Establish and maintain software development files (SDFs).	2167A, 4.1.8 2167A, 4.2.9 973, 5.3.3.3
3. Establish software and documentation libraries.	973, 5.3.3.1 973, 5.3.3.3
4. Provide and control access to development histories.	2167A, 4.4.3 973, 4.3.1 973, 4.3.3
5. Prepare and maintain management records, status reports, and software evaluation records.	2167A, 4.4.3 2167A, 4.5.3
6. Analyze configuration status accounting data.	973, 5.5.7
7. Record the current, approved software, documentation, and identifiers.	973, 4.3.2.b 973, 4.6.a 973, 5.3.5 973, 5.5.4 973, H.5.1.1.1 973, H.5.1.1.2 973, H.5.1.1.5 973, H.5.1.1.6 973, H.5.1.1.7 973, H.5.1.1.8
8. Record and report the status of requests for engineering changes, deviations and waivers.	973, 4.6.b 973, 4.6.d 973, 5.5.4 973, H.5.1.2
9. Record and report implementation status of authorized changes.	973, 4.6.e 973, 4.6.f 973, 5.3.3 973, 5.5.4 973, 5.5.8 973, H.5.1.3 973, H.5.1.4 973, H.5.1.5.1 973, H.5.1.5.3

<u>Configuration Status Accounting Requirements (cont.)</u>	<u>Reference</u>
10. Record and report the location of each CSCI version in the field.	973, 4.6.g 973, H.5.1.6.1.2
11. Ensure information about new releases is incorporated into the configuration status accounting system.	973, 5.3.5.2.1 973, 5.3.5.2.2
12. Record and report the results of configuration audits.	973, 4.6.c 973, 5.6.2.1 973, 5.6.2.4.b 973, 5.6.3.1 973, H.5.1.7

Appendix B: Software Configuration Management (SCM) Tool Vendor List

Software Configuration Management (SCM) Tool Vendor List

Tool	Point of Contact	Vendor/Address
Clear Case	Ms. Norma R. McCluskey Tel: 508-650-1193 ext. 39 Fax: 508-650-1196 email: norma@atria.com	Atria 24 Prime Park Way Natick, MA 01760
Ensemble	Mr. Ron Imbriale Tel: 401-351-2273 Fax: 401-351-7380 email:	Cadre 222 Richmond St. Providence, RI 02903
CaseWare/CM	Mr. Riz Haq Tel: 214-392-3008 Fax: 214-960-9911 email: rizh@cwi.com	CaseWare, Inc. 14785 Preston Rd, #550 Dallas, TX 75240
DOMAIN Software Engineering Environment (DSEE)	Mr. Mike Gallagher Tel: 800-752-0900 Fax: email:	Hewlett Packard Company 5301 Stevens Creek Blvd. PO Box 58059, MS 51LSG Santa Clara, CA 95052-8059
Rational Control	 Tel: 408-496-3600 Fax: 408-496-3636 email:	Rational 3320 Scott Blvd. Santa Clara, CA 95054-3197
Aide-De-Camp (ADC)	Ms. Susan Paquet Tel: 508-369-7398 Fax: 508-369-8272 email: adc@smds.com	Software Maintenance & Development Systems, Inc. 200 Baker Ave., Suite 300 Concord, MA 01742
Product Configuration Management System (PCMS)	Mr. Sohail Haque Tel: 703-760-7895 Fax: 703-760-7899 email:	SQL Software Ltd. 8000 Towers Crescent Dr., Suite 1350 Vienna, VA 22182
TeamTools	Mr. Gary Wilkins Tel: 408-730-3500 Fax: 408-730-3510 email: sun.com!teamone!gary	TeamOne Systems, Inc. 710 Lakeway Dr., Suite 100 Sunnyvale, CA 94086
Source_Manager	Ms. Virginia K. Jones Tel: 408-227-7700 Fax: 408-227-7757 email:	TransWare Enterprises, Inc. 5450 Thornwood Dr., Suite M San Jose, CA 95123-1222

Appendix C: Software Configuration Management (SCM) Tool Evaluation

Aide-De-Camp (ADC)

Software Configuration Management (SCM) Tool Evaluation

Tool Name:	Aide-De-Camp (ADC)
Version Number:	• ADC v8.01.2 • CM Model 209 v3.0.10 (A utility program consisting of macros and scripts to simplify ADC operation.) • X-ADC v1.0 (A graphical user interface.)
Release Date:	July 1993
Frequency of Updates:	Semiannual version releases with intermittent bug fixes
Date of First Release:	1983
Number Sold:	over 300 licenses (both individual and site licenses)
Vendor:	Software Maintenance & Development Systems, Inc.
In Business Since:	1981
Address:	200 Baker Avenue, Suite 300 Concord, MA 01742
Point of Contact:	Susan R. Paquet
Phone Number:	508-369-7398
FAX Number:	508-369-8272
Email Address:	adc@smds.com

Platforms/Operating Systems: DEC VMS, DEC ULTRIX, IBM RS/6000, HP HP/UX, HP 9000, MIPS, SPARC, Apple A/UX, 386/486, and Silicon Graphics

Programming Languages Supported: Ada, C, FORTRAN

Description: Aide-De-Camp is an object-oriented tool (versus file oriented) utilizing a relational database, which represents changes logically as well as physically. ADC captures a change as an object (a grouping of logically related entities associated with the particular change): source modules, documentation, design diagrams, and other information. This group of logically related entities is defined as a change set (*cset*); change sets have attributes that describe the change logically and properties that describe the change physically (i.e., changed source code lines). Versions are simply a collection of change sets specified by the user. ADC supports simultaneous access to files, multiple projects, merging development paths, conflict detection, automated configuration builds, and database security.

SCM Requirements-Functionality Evaluation Matrix: (attached)

Substantiation of SCM Requirements met by Tool Functionality:

I. Configuration Identification

1. Document and implement plans for performing configuration identification. ADC can store an ASCII text file in the database. It is simply stored as an object of type *source* with a specific file name that identifies it as a configuration identification plan for SCM. ADC provides a standard set of macros and the capability to define user-specific macros to aid in the identification and tracking of source code, object code, documentation, and other information requiring identification for the effort. Because each macro has a defined usage, ADC can help define the identification procedures for a given project. The system administrator will define all the pertinent objects that need to be identified, any attributes

which help describe and identify each instance of an object, and the relationships or associations between the objects.

2. Select CSCIs. This is a manual task and is not supported by ADC. Once selected, ADC will record, track, and report CSCI information. See requirement I-3 immediately following.

3. For each CSCI, identify baselines, developmental configuration, and associated documentation. Each CSCI can be defined as an object with the attribute "csets" that identify a specific group of *csets*. Each *cset* is associated with a source file containing documentation, plans, etc. in the form of ASCII text. Formal baselines are established when a version or group of *csets* is *installed* using ADC's *checkpoint* feature. Once a version is *installed* it cannot be changed, it is frozen. When *installed* is performed a duplicate copy of the version is created which can be changed and is considered *plastic*. The CSCI's developmental configuration is the *plastic* version of the version *installed* after the Software Specification Review, when the allocated baseline is established.

4. Trace CSCIs to WBS elements when MIL-STD-881 is invoked. As stated previously, CSCIs can be defined as objects. In addition to the cset attribute CSCI objects can have an attribute "WBS" which specifies which Work Breakdown Structure elements a given CSCI pertains. Likewise, an object can be defined as "WBS elements" with the attribute "CSCI". This provides a two way link between each WBS element and the applicable CSCI.

5. Decompose each CSCI into CSCs and CSUs. ADC supports a package that establishes and tracks hierarchies. Hierarchies can be set up for any entity type. To meet this requirement, source files would be the entities of a source hierarchy resembling a CSCI, CSC, and CSU breakout. The user must manually define the hierarchy using the *defhier* macro command. This macro establishes the "ultimate ancestor" or highest parent in the hierarchy representing a CSCI. Utilizing the *addhier* macro which adds a parent-child pair, source files representing CSCs can then be added. The same procedure applies for representing CSUs. Once the initial hierarchy is established the user can maintain it using ADC's macro commands such as reporting the full contents of a hierarchical tree in an indentured listing and deleting a parent-child pair.

6. Identify and document the version of each SCI corresponding to the documentation. ADC will automatically assign a name for each file it maintains. If this is not appropriate the user may assign his own naming scheme unless Model 209 is used. The vendor states that Model 209's process enforcement requires the automatic assignment of names to *csets*. However there are ways around such constraints. Each SCI is managed as a source object consisting of a file containing source code. This object can have a user-defined object attribute such as "ID" or "CPIN" depending on the requirements of the contract. The documentation is defined as a source object consisting of files containing ASCII text. These objects can also have a user-defined object attribute such as "ID" or "CPIN." These two object classes can be associated with each other through attributes. The source code objects have an attribute "documentation" that specifies specific documentation associated with a specific file containing source code. Likewise, the documentation objects have an attribute "source code" that specifies source code described by a particular document. Documenting the exact version of each SCI corresponding to a particular document is facilitated by ADC's report capabilities. A query of all source code attributes for each instance of (or specific) documentation objects can be performed and reported as a list, which can then be stored in an ASCII text file. This file then becomes a new instance of a document object, and can be easily updated by reissuing the *report to a list* command.

7. Identify, define, and document interfaces. ADC automatically creates a dependency attribute for each change set object or source file. This attribute identifies which software modules call and are called by the software modules contained in the change set object or source file. The definition and documentation of each interface is a manual task for the user but, once completed, ADC will track and control this documentation as ASCII text files stored as source objects. The source objects, consisting of files containing source code, can have an attribute defined as interfaces that associates the source code to its interface requirements. ADC will maintain this association to facilitate tracking and control of source code and documentation concurrently. See the discussion for requirement I-9 below.

8. For each CSCI, allocate and provide traceability for requirements to lower SCIs and documentation. ADC does not have the capability to allocate requirements. This is a manual task left up

to the system administrator. ADC does provide traceability between requirements, SCIs, and documentation although the vendor states that this is not the intended purpose of the tool The method is similar to the discussion in requirement I-4 above dealing with tracing CSCIs to WBS elements. In this case a requirements object is defined. This object has the attributes "documentation" and "source code." This associates the documentation and source code objects to the requirements object. The documentation object has the attributes "source code" and "requirements." Likewise, the source code object has the attributes "requirements" and "documentation". In this manner, a specific requirement can be traced to specific source code and document(s), a particular document to specific requirement(s), and given source code to the requirement(s) it is driven by.

9. Ensure correlation between each SCI, its documentation, and other associated data. ADC tracks changes by logically grouping all entities related to a change. This includes source code, documentation, and other data. ADC requires that the user designate a file which documents a change that the user wishes to *check in* to an alpha, beta, or release version. In addition, ADC automatically includes the file in the database, tags the file as documenting the change, and verifies that the file is not empty before ADC actually processes the change. This ensures that a given version of software includes the correct supporting documentation from which it was developed, as well as any other supporting data. It will not, however, guarantee that the documentation is correct, but only that a specific source code cannot be selected which would inadvertently select the wrong supporting documentation. When a user selects a specific version by identifying a collection of change sets, ADC automatically links the appropriate entities and builds the configuration.

10. Display information about an identifier upon command. The ADC system stores and tracks information contained in the database as objects. Each object is defined as a specific type, such as a *cset* (change set), *directory*, or *source* file. ADC manages any type of data or information as objects with specific attributes. Identifiers may be defined as objects of the type *attribute* or *abstract*. These "identifier objects" can then be associated with objects of the type *source* or *cset*. When the source file or cset is installed (baselined), so is the identifier – and neither can be changed. ADC provides a querying facility through the *display* command that will retrieve a particular instance of an identifier object, then report it to the CRT screen along with its information.

II. Configuration Control

1. Document and implement plans for performing configuration control. ADC can store an ASCII text file in the database. It is simply stored as an object of type *source* with a specific file name that identifies it as an configuration control plan for SCM. ADC contains a macro command language that has many standard and tailorable commands. In addition, reports can be created to a user's specific needs. The system administrator for ADC can develop macros and special reports/form to be used in performing configuration control for those functions which are not done automatically by ADC. This in effect defines and enforces many of the procedures to be used by the developers on a given project. However, the process of defining a plan, and its associated procedures, is still a manual task and is left up to the user.

2. Establish an engineering release system. ADC/CM Model 209 provides a generic engineering system called the M209 and is based on four development phases: development, alpha test, beta test, and customer release. This supports an engineering release system in that the only software and documentation formally distributed to a customer is that which has successfully reached the customer release phase. No products should be released to a customer while in any other phase. The system administrator for ADC or the project leader has the capability to *install* a version when the software has been thoroughly tested and the documentation has been reviewed and both are considered stable. This does not completely meet the intent of this requirement because all that is really being accomplished is identifying and preventing changes to a tested and verified version. Levels of approval, sign off authority, and release procedures are left up to the user to define. The user can define his own process using the ADC software without using the M209.

3. Document and implement a corrective action process. Same methodology and concepts described in requirement II-1 above.

4. Apply internal configuration control prior to baselining products. With ADC, versions are considered to exist in two distinct states, installed (baselined) and plastic (under development). An installed version cannot be changed whereas a plastic version may be modified as development progresses. A version is installed using an option called *checkpoint* that freezes software, documentation, and other associated data at any time. The project leader does not necessarily only use *checkpoint* when establishing a baseline. The user can freeze files at any time (e.g., when a CSU has been successfully coded and tested). When a *checkpoint* is performed, ADC will create a working copy of the version to be frozen so that development towards the next version can begin. This working copy is initially the same as the frozen version, except that the version number has automatically been incremented by one unit so that historical development can be maintained. ADC supports four development phases: development, alpha test, beta test, and customer release. Alpha test is in-house testing by those individuals involved in the development of the software. Beta test involves customer testing, and customer release is the final version of the software, approved for customer use. These are considered contractor internal baselines, and with the exception of the development phase, changes are considered "bug fixes."

5. Maintain master copies of, and control changes to, deliverable software and documentation. As mentioned in requirement II-4 above, by using *check point*, any software, documentation, etc., can be frozen from change to establish a protected baseline. No changes can be made to information contained in files which have been *installed* using *check point*.

6. Prepare a problem/change report for each problem detected. ADC provides various pre-defined reports and has the capability to tailor reports. Using the commands: *openfile*, *writefile*, and *closefile*, the user-defined reports can be created. A series of ADC macro commands can be developed that will query the database, retrieve the requested data, and format the output in the form required by a user-defined report. Initially, problem reports must be manually entered into the database by the system administrator as a problem_report object. This object can then be assigned descriptive attributes such as identification, priority_level, status, and assigned_to. The vendor agrees that ADC has the capability to track "bugs", but the vendor recommends purchasing an additional tool that can interface with ADC and is more adequate for the purpose.

7. Prepare and classify changes to baselined documentation and software. As stated for requirement II-6 above, reports can be tailored to meet the specific needs of the user, such as reporting an engineering change proposal in the DD Form 1692 format. In addition, through the use of *csets* ADC logically groups all entities or objects associated with a particular function or change. For example, consider a change needed to incorporate a new software interface. All necessary software and documentation changes, engineering change proposal, and other data required to incorporate this new interface will be grouped as one unique *cset*. A version is seen by ADC as a group of particular *cset*. A change to an existing version can be accomplished by developing a new *cset* and adding it to the group of *csets* to create a new version. Since a *cset* is a unique entity, information about a particular change can be retrieved rather easily. The classification can be merely an attribute of a *cset* or a *keyword* object.

8. Provide access to documentation and code under control. Through ADC's CM Model 209 security features, access to the database is controlled. ADC provides for three authorization levels: system administrator, project leader, and developer. The administrator has access to all of the ADC functions and databases; a project leader has access to his or her project database, and can establish baselines by freezing software, while also performing configuration builds. A developer can *checkin* and *checkout* files for development within a given project. Only the administrator can assign access authority to individuals, and only one authorization level is assigned per individual, no matter how many projects each is assigned. Government individuals can be issued an authorization level to access the database. Modification to Model 209 can be performed where attributes can be set to flag a user as "read only". Such users would be permitted to access file but not to perform the Checkout capability, thus not able to modify files. In addition, ADC supports concurrent access to a version, phase, release of software. Access to any given software, documentation, or other data will not be prevented if the requested data is already being accessed by someone else.

III. Configuration Auditing

1. Provide information in support of formal reviews and audits. Using ADC's *report* feature, any information in the database can be retrieved and reported. Much of the information necessary to support an audit or review can be reported by ADC. For example, an *installed* (baselined) version can be reported, along with all objects associated or having a relationship with that version. This would include individual files containing source code or documentation, change history, and lists of change requests, to name a few.

2, Participate in the resolution of discrepancies identified during reviews and audits. While ADC does not actively resolve discrepancies, it can record them; they can be defined as a "discrepancy" object of type *source*, or even *generic*. *Source* is a better choice since *generic* objects cannot be installed and are therefore changeable. Each instance of a discrepancy can be associated with one or more other objects, such as *source* objects. In this way a file containing source code is linked to the discrepancy objects (files containing discrepancy information). If the object is of type *source* and has been installed, a history of a specific discrepancy can then be developed. However, ADC does not provide a tracking capability that can actively monitor the scheduled close-out date and the progress to date. This information must be manually input to, and monitored by, the user.

3. Record and publish meeting minutes. ADC facilitates the storing of ASCII files, such as minutes from a Functional Configuration Audit. This is accomplished by defining "minutes" as an object of type *source*. The file containing this text information can be displayed on (reported to) a CRT screen, or printed on hard copy if a printer is installed on the hardware platform supported by ADC.

IV. Configuration Status Accounting

1. Document and implement plans for performing configuration status accounting. ADC can store an ASCII text file in the database, simply as an object of type *source* with a specific file name that identifies it as a configuration status accounting plan for SCM. ADC provides a standard set of macros, and the capability to define user specific macros to assist in performing status accounting of information required for the effort. Because each macro has a defined usage, ADC can help define the status accounting procedures for a given project. The system administrator will define all the pertinent information and data that need to be recorded, stored, and reported.

2. Establish and maintain software development files (SDFs). ADC utilizes an integrated database, and can archive and track any ASCII and non-ASCII files, as well as source and object code. ADC normally stores information about an entire project's software and documentation in a single database, separate from the database used for SCM of the development of the project. This provides an additional level of security and ensures the integrity of the development history by removing the database from direct user access. Transfer commands provided by ADC allow the development information to be exported to the historical database. As just stated, ADC stores historical information for the entire project in a single database. This is somewhat different than a hard copy version of a software development file, which is usually an individual file per unit of software. Having a single database is not a problem since ADC uses object-oriented methodology. In this way, all related entities are grouped logically and can be retrieved easily. Although the information about all software in the project is stored together, everything is stored as objects with associations; information pertaining to specific software can be retrieved using an *import* command.

3. Establish software and documentation libraries. Files containing source code, object code, and documentation are stored in a repository within the ADC database. Using the various query and list commands provided by ADC, software and documentation can be retrieved from the database much like that from a library. This is a simple case of database management.

4. Provide and control access to development histories. Histories are automatically tracked as each SCI is developed. Source code, documentation, and other text information are defined as objects of type *source* which can be installed (baselined) using the *checkpoint* feature. To make a change to an installed source object a new version consisting of one or more change sets (cset) is created. ADC tracks these csets, maintaining a history of development from the initial version through all csets based on that initial version. This history is stored in the database. As described earlier, ADC provides a security

facilities which controls access to the database. The system administrator specifies who has access and what level of access. ADC records all system transactions to provide an audit trail of system accesses.

5. Prepare and maintain management records, status reports, and software evaluation records. ADC can create custom reports using loops, format statements, and conditional testing, supported by a system macro language. All information contained in the ADC database is accessible, and can be presented in a user defined report. Since, as already stated, ADC is capable of storing any ASCII text file, these reports and records can be stored and maintained within the ADC database.

6. Analyze configuration status accounting data. ADC provides no capability to automatically examine problem reports in order to determine a trend and to verify if problem trends have been eliminated. This can be manually accomplished, using various query commands provided by ADC. The database can be queried for all instances of problem reports concerning problem X. ADC will retrieve this information, but it is up to the user to analyze the data and determine if the trend is no longer present. For this reason, ADC does not meet this requirement.

7. Record the current approved software, documentation, and identifiers. This capability is inherent in ADC since it is built around a relational database purposefully developed for storing information about a project, such as software and documentation. ADC allows the user to freeze software at a specific point in development (e.g., the establishment of the allocated baseline). The software, documentation, and other related data is archived and cannot be changed inadvertently.

8. Record and report the status of request for engineering changes, deviations, and waivers. The same methodology and capabilities discussed for requirement II-6 earlier are also applicable to this requirement. Requests for changes, deviations, or waivers prepared in this requirement can be recorded and tracked in the same way as problem reports.

9. Record and report implementation status of authorized changes. See requirement II-7 above. Authorized changes are grouped logically as a *cset*. This *cset* can have an attribute that identifies the status of the change, in addition to other attributes that further describe the *cset*. *Csets* can be stored just like any software file for historical purposes (see IV-2).

10. Record and report the location of each CSCI version in the field. Information concerning the location, or sites, using a particular version of software is stored in a data file and is treated as an object, just like the documentation for that software. Software versions are automatically linked to the documentation, site, and any other data files. A query of a specific software version will retrieve, through the relational database, which sites currently use that version. A query can also be from the opposite direction, determining which version a specific site uses.

11. Ensure information about new releases is incorporated into the status accounting system. Products (software, documentation, etc.) are stored in, checked out from, checked into, and released from the ADC database. Therefore information about a version is gathered all through development, testing, and release. Information concerning new and old releases is archived in a historical database for future use.

12. Record and report the results of configuration audits. ADC archives and retrieves ASCII files, such as minutes from a Functional Configuration Audit. The file can be displayed (reported to) a CRT screen or printed on hard copy, if a printer is installed on the hardware platform supported by ADC. See requirement II-6 above regarding the generation of user-defined reports.

Comments: Aide-De-Camp's key strength is its ability to relate various files together whether they be source code, documentation, manuals, or reports. This permits a project to be ordered logically in a way that makes software configuration management more effective.

Tool Name: Aide-De-Camp • Vendor:
Software Maintenance & Development Systems, Inc.

	Database Management	Configuration Build	Decomposition/development Control	Work Area Control	Change Control	Baseline Management	Customization
I. CONFIGURATION IDENTIFICATION							
1. Document and implement plans for performing configuration identification.	X						X
2. Select CSCIs.							
3. For each CSCI, identify baselines, developmental configuration, and associated documentation.	X			X		X	
4. Trace CSCIs to WBS elements when MIL-STD-881 is invoked.	X		X				
5. Decompose each CSCI into CSCs and CSUs.	X		X				
6. Identify and document the version of each SCI corresponding to the documentation.	X						X
7. Identify, define, and document interfaces.	X	X					
8. For each CSCI, allocate and provide traceability for requirements to lower SCIs and documentation.	X		X				
9. Ensure correlation between each SCI, its documentation, and other associated data.		X		X	X		
10. Display information about an identifier upon command.	X						
II. CONFIGURATION CONTROL							
1. Document and implement plans and procedures for configuration control.	X				X		X
2. Establish an engineering release system.						X	
3. Document and implement a corrective action process.	X				X		X
4. Apply internal configuration control prior to baselining products.			X			X	
5. Maintain master copies of, and control changes to, deliverable software and documentation.						X	
6. Prepare a problem/change report for each problem detected.	X						X
7. Prepare and classify changes to baselined documentation and software.	X	X			X		X
8. Provide access to documentation and code under configuration control.	X			X			
III. CONFIGURATION AUDITING							
1. Provide information in support of formal reviews and audits.	X	X					
2. Participate in the resolution of discrepancies identified during reviews and audits.	X						
3. Record and publish meeting minutes.	X						X
IV. CONFIGURATION STATUS ACCOUNTING							
1. Document and implement plans and procedures for performing configuration status accounting.	X						X
2. Establish and maintain software development files (SDFs).	X						
3. Establish software and documentation libraries.	X						
4. Provide and control access to development histories.	X			X	X		
5. Prepare and maintain management records, status reports, and software evaluations records.	X						X
6. Analyze configuration status accounting data.							
7. Record the current, approved software, documentation, and identifiers.	X						
8. Record and report the status of request for engineering changes, deviations, and waivers.	X						X
9. Record and report implementation status of authorized changes.	X	X			X		X
10. Record and report the location of each CSCI version in the field.	X						
11. Ensure information about new releases is incorporated into the status accounting system.	X					X	
12. Record and report the results of configuration audits	X						X

Appendix D: Software Configuration Management (SCM) Tool Evaluation

Product Configuration Management System (PCMS)

Software Configuration Management (SCM) Tool Evaluation

Tool Name:	Product Configuration Management System (PCMS)
Version Number:	3.3
Release Date:	16 July 1993
Frequency of Updates:	semiannually
Date of First Release:	1988
Number Sold:	
Vendor:	SQL Software Ltd.
In Business Since:	1987
Address:	8000 Towers Crescent Dr. Suite 1350 Vienna, VA 22182
Point of Contact:	Mr. Steven F. X. Murphy
Phone Number:	703-760-7895
FAX Number:	703-760-7899
Email Address:	

Platforms/Operating Systems: VMS, ULTRIX, and UNIX operating systems; portable to Digital, SUN, BULL, Sequent, Hewlett Packard, and ICL platforms

Programming Languages Supported: Ada, C, Pascal, FORTRAN, and ASM

Description: PCMS is an active CM tool which combines integrated change management, an automated configuration build facility, a life cycle and role management engine, a release manager, and product design and development modules with the full support of an open relational database. PCMS supports the development, production, and maintenance cycles of the hardware, software, and documentation items of a product.

SCM Requirements-Functionality Evaluation Matrix: (attached)

Substantiation of SCM Requirements met by Tool Functionality:

I. Configuration Identification

1. Document and implement plans for performing configuration identification. PCMS's customization functionality permits the user to input rules, plans, and formats governing the various PCMS facilities in order to meet company standards, procedures, and products. For instance, the user can specify valid types of *product-items* that will be used to implement the product.

2. Select CSCIs. Although PCMS does not directly select CSCIs for the user, its decomposition/development control functionality facilitates the user's effort to decompose the product into a hierarchy of *design-parts* which, in turn, are implemented as *product-items* (i.e. CSCIs). Any form of information that can reside on a disk (i.e. software source or object code, specification documents, meeting minutes, change tracking forms, hardware control documents, etc.) can be implemented as a *product-item* and managed as an object within PCMS's relational database system.

3. For each CSCI, identify baselines, developmental configuration, and associated documentation. PCMS's baseline management functionality allows for a snapshot of part, or all, of the product to be taken at any moment, thus preserving the compatible *design-parts* and *product-items* associated with it. PCMS provides two forms of baselines: a *design-baseline* and a *release-baseline*. The

design-baseline includes parts and items which are not frozen, and therefore emulates the development configuration, whereas the *release-baseline* captures details for a specific configuration or release of the product.

4. Trace CSCIs to WBS elements when MIL-STD-881 is invoked. PCMS's database management and customization functionalities give the user complete flexibility in defining not only the objects that will reside in the database, but also the attributes of each object. Therefore, CSCIs can be traced to WBS elements by defining a WBS identification attribute for each object.

5. Decompose each CSCI into CSCs and CSUs. PCMS provides for the product to be decomposed into a hierarchy of *design-parts*. *Design-parts* can emulate CSCIs, CSCs, and CSUs. Each *design-part* is then implemented by one or more *product-items* (e.g. specification documents, hardware, software).

6. Identify and document the version of each SCI corresponding to the documentation. PCMS's database management and decomposition/development control functionalities provide for each object (i.e., *design-part, product-item*) to be identified by both a part specification and an item specification. The attributes associated with each of these specifications uniquely identify each SCI related to the overall product and also are used to define relationships between the SCIs and the design hierarchy.

7. Identify, define, and document interfaces. PCMS's decomposition/development control functionality ensures that associations (to include interfaces) between design-parts and product-items are identified and defined as *product-items* and are implemented. PCMS's relational database structure is modeled on these and other established relationships. Interfaces between SCIs can also be documented in Interface specifications (implemented as *product-items*), which can be based upon information obtained from querying the repository/libraries.

8. For each CSCI, allocate and provide traceability for requirements to lower SCIs and documentation. PCMS's decomposition/development control functionality ensures that as the product evolves, it is broken down into a hierarchy of *design-parts* which encapsulate all product functions or requirements. Implementation of *design-parts* into *product-items*, and the corresponding establishment of relationships and associations, ensure that the requirements are traceable.

9. Ensure correlation between each SCI, its documentation, and other associated data. PCMS's database management and decomposition/development control functionalities ensure that all interrelationships between *design-parts* and *product-items* are documented in the repository's, or libraries', relational database structure.

10. Display information about an identifier upon command. PCMS's database management functionality, which is based on a object-oriented relational database structure, permits the user to query the repository or libraries to obtain information about any object (*design-part* or *product-item*). Upon request, PCMS will display any attribute information related to an identifier.

II. Configuration Control

1. Document and implement plans and procedures for configuration control. PCMS's customization functionality permits the user to input rules, plans, and formats governing the various PCMS facilities in order to meet company standards, procedures, and products. For instance, the user can specify both the valid types of *change-documents* that will be used to identify, track, and implement changes to the product, and the procedures that will be followed during the change process.

2. Establish an engineering release system. PCMS's database management functionality provides for libraries to be created in which *product-items* can be stored. PCMS gives the customer the flexibility to store each type of *product-item* (i.e., source code, change documents, specifications, etc..) in a separate library, or group *product-item* types together in one or more libraries. In any event, libraries are created before work is started on the product.

3. Document and implement a corrective action process. PCMS's change control and customization functionalities enable the user to define the change/enhancement format and procedures to be used in identifying, tracking, and implementing necessary changes and/or enhancements to the product.

4. Apply internal configuration control prior to baselining products. PCMS's baseline management functionality permits a *design-baseline* to be established consisting of the *design-parts* and *product-items* associated with a portion of the product. Because the parts and items associated with a *design-baseline* are not frozen, this type of baseline scheme can be used to emulate the developmental configuration, or the company's internal configuration for a product under development.

5. Maintain master copies of, and control changes to, deliverable software and documentation. PCMS's baseline management functionality ensures that the component versions included in a release (i.e., deliverable) baseline can not be deleted or amended. Changes to such components can be made, but PCMS ensures that these changes are recorded as new versions so that the *release-baseline* (i.e., master copy) is preserved.

6. Prepare a problem/change report for each problem detected. PCMS's change control and customization functionalities permit the user to customize the types of *change-documents* to be used and the processes to be followed, and thereby implement a problem/change reporting system.

7. Prepare and classify changes to baselined documentation and software. PCMS's change control and customization functionalities permit the user to create multiple *change-document* types. Therefore, changes can be classified by formatting a different type of *change-document* for each change classification area (i.e., Class I ECP, Class II ECP, SCN; critical, major, and minor waivers and deviations).

8. Provide access to documentation and code under configuration control. PCMS's work area control and customization functionalities provide for a scheme to both provide and restrict access to *product-items* under control. The user can assign *roles* (e.g., designer, developer, reviewer, etc.) to members of the product team, and can also define types of *lifecycles* for each and every *product-item*. Based on this user-defined control information, PCMS controls access to all *product-items* contained in the repository/libraries by ensuring that a team member has the correctly assigned *role*, given the *lifecycle* state of a particular *product-item* to access that item.

III. Configuration Auditing

1. Provide information in support of formal reviews and audits. PCMS's database management functionality permits the user to query the repository for information in support of reviews and audits. Such information might include a list of outstanding action items, source code listings, and/or meeting minutes. PCMS's configuration build functionality permits the user to build various configurations of the product for both test and official release purposes.

2. Participate in the resolution of discrepancies identified during reviews and audits. As substantiated in II-3 above, PCMS's change control and customization functionalities provide a scheme for the identification, tracking, and resolution of discrepancies, regardless of whether they occur during coding, testing, or auditing.

3. Record and publish meeting minutes. PCMS's database management functionality permits any form of data that can reside on a disk (i.e., meeting minutes) to be implemented as an object (product-item), and managed within the repository/libraries.

IV. Configuration Status Accounting

1. Document and implement plans and procedures for performing configuration status accounting. Same substantiation as used for II-1 above.

2. Establish and maintain software development files (SDFs). PCMS's database management functionality permits the user to establish and manage object relationships. In this manner, all product-items related to the same *design-part* can be accessed, and their output managed within a physical filing system.

3. Establish software and documentation libraries. Same substantiation as used for II-2 above.

4. Provide and control access to development histories. PCMS's work area control and customization functionalities provide for a scheme to both provide and restrict access to *product-items* under control. The user can assign *roles* (e.g., designer, developer, reviewer, etc.) to members of the

product team and define types of *lifecycles* for each and every *product-item*. Based on this user defined control information, PCMS controls access to all *product-items* contained in the repository/libraries by ensuring that a team member has the correctly assigned *role*, given the *lifecycle* state of a particular *product-item* to access that item.

5. Prepare and maintain management records, status reports, and software evaluation records. PCMS's customization functionality permits the user to define *product-item* types (i.e., evaluation results document) and then implement any such occurrence of that type as a *product-item*, and thus associate it with any applicable *product-items* and *design-parts*. PCMS's database management functionality then permits the user to maintain that object within the repository/libraries.

6. Analyze configuration status accounting data. PCMS's database management and customization functionalities enable the user to define different types of *report-documents*. In these *report-documents*, the user can define selection rules so that, when the document is implemented, it automatically queries the database, compiles information, and integrates this information in the report. In this fashion, status accounting data, such as discrepancy report trends, can be automatically compiled, analyzed, and reported upon.

7. Record the current, approved software, documentation, and identifiers. PCMS's baseline management functionality provides for a means to capture the *design-structure* and all associated *product-items* at any point in time. A *release-baseline* can be used to capture the *design-parts* and *product-items* of the current approved configuration.

8. Record and report the status of requests for engineering changes, deviations, and waivers. As substantiated in II-3 above, PCMS's change control and customization functionalities provide a means for engineering changes, deviations and waivers to be implemented as *change-documents*. PCMS's database management functionality permits the user to make interactive queries concerning the status of these or any other object types.

9. Record and report implementation status of authorized changes. Same substantiation as used for IV-8 above.

10. Record and report the location of each CSCI version in the field. PCMS's baseline management functionality provides the user with release control provisions to record details pertaining to which customers have received which release baseline configurations.

11. Ensure information about new releases is incorporated into the status accounting system. Same substantiation as used in IV-10 above.

12. Record and report the results of configuration audits. Same substantiation as used in III-3 above.

Comments: The key strengths of PCMS appear to be its ability to manage the configuration of both the software and hardware items comprising a product, to manage all product components as objects within a relational database structure, and to openly interface and exchange data with other tools in the user's software engineering environment.

Tool Name: Product Configuration Management System (PCMS)
Vendor: SQL Software Ltd.

	Database Management	Configuration Build	Decomposition/development Control	Work Area Control	Change Control	Baseline Management	Customization
I. CONFIGURATION IDENTIFICATION							
1. Document and implement plans for performing configuration identification.							X
2. Select CSCIs.							
3. For each CSCI, identify baselines, developmental configuration, and associated documentation.	X		X				X
4. Trace CSCIs to WBS elements when MIL-STD-881 is invoked.	X						X
5. Decompose each CSCI into CSCs and CSUs.			X				
6. Identify and document the version of each SCI corresponding to the documentation.	X		X				
7. Identify, define, and document interfaces.	X		X				
8. For each CSCI, allocate and provide traceability for requirements to lower SCIs and documentation.			X				
9. Ensure correlation between each SCI, its documentation, and other associated data.	X		X				
10. Display information about an identifier upon command.	X						
II. CONFIGURATION CONTROL							
1. Document and implement plans and procedures for configuration control.							X
2. Establish an engineering release system.	X						
3. Document and implement a corrective action process.					X		X
4. Apply internal configuration control prior to baselining products.						X	
5. Maintain master copies of, and control changes to, deliverable software and documentation.	X						
6. Prepare a problem/change report for each problem detected.					X		X
7. Prepare and classify changes to baselined documentation and software.					X		X
8. Provide access to documentation and code under configuration control.				X			X
III. CONFIGURATION AUDITING							
1. Provide information in support of formal reviews and audits.	X	X					
2. Participate in the resolution of discrepancies identified during reviews and audits.					X		X
3. Record and publish meeting minutes.	X						
IV. CONFIGURATION STATUS ACCOUNTING							
1. Document and implement plans and procedures for performing configuration status accounting.							X
2. Establish and maintain software development files (SDFs).	X						
3. Establish software and documentation libraries.	X						
4. Provide and control access to development histories.				X			X
5. Prepare and maintain management records, status reports, and software evaluations records.	X						X
6. Analyze configuration status accounting data.	X						X
7. Record the current, approved software, documentation, and identifiers.						X	
8. Record and report the status of request for engineering changes, deviations, and waivers.	X				X		X
9. Record and report implementation status of authorized changes.	X				X		X
10. Record and report the location of each CSCI version in the field.						X	
11. Ensure information about new releases is incorporated into the status accounting system.						X	
12. Record and report the results of configuration audits	X						

Bibliography

Ambriola, Vincinzo, L. Bendix, and P. Ciancarini. "The Evolution of Configuration Management and Version Control," *Software Engineering Journal,* 303-310 (November 1990).

ANSI/IEEE Standard 828-1990. *IEEE Standard for Software Configuration Management Plans*. New York: The Institute of Electrical and Electronics Engineers, Incorporated, 1990.

ANSI/IEEE Standard 1042-1987. *IEEE Guide to Software Configuration Management.* New York: The Institute of Electrical and Electronics Engineers, Incorporated, 1988.

Babich, Wayne A. *Software Configuration Management: Coordination for Team Productivity.* Reading MA: Addison-Wesley Publishing Company, 1986.

Berlack, H. Ronald. *Software Configuration Management.* NewYork: John Wiley and Sons, Incorporated, 1992.

Bersoff, Edward H. "Elements of Software Configuration Management," *IEEE Transactions on Software Engineering,* 10: 79-87 (January 1984).

---- and Alan M. Davis. "Impacts of Life Cycle Models on Software," *Communications of the ACM,* 34: 105-118 (August 1991).

----, Vilas D. Henderson, and Stan G. Siegel. *Software Configuration Management: An Investment in Product Integrity.* Englewood Cliffs NJ: Prentice-Hall, Incorporated, 1980.

----, Vilas D. Henderson, and Stan G. Siegel. "Attaining Software Product Integrity," *Tutorial: Software Management.* 341-348. New York: IEEE Press, 1981 (EH0189-1).

----. "Software Configuration Management: A Tutorial," *Tutorial: Software Configuration Management.* 24-32. New York: IEEE Press, 1980 (EHO 169-3).

Brown, Bradley J. "Checksum Methodology as a Configuration Management Tool," *Journal of Systems and Software*, 7: 141-143 (1987).

Dean, William A. "Why Worry About Configuration Management?" *Defense Systems Management Review*, 2: 21-29 (Summer 1979).

Department of Defense. *Configuration Management.* MIL-STD-973. Washington: DoD, 17 April 1992.

Department of Defense. *Defense System Software Development.* DoD-STD-2167A. Washington: DoD, 29 February 1988.

Department of Defense. *Defense System Software Quality Program.* DoD-STD-2168. Washington: DoD, 29 April 1988.

Department of Defense. *Mission Critical Computer Resources Management Guide*. Ft. Belvoir VA: Defense Systems Management College, no date.

Feiler, Peter H. "Configuration Management Models in Commercial Environments," *CMU/SEI-91-TR-7*, Pittsburgh PA: Software Engineering Institute (SEI), Carnegie Mellon University, March 1991.

Ferens, Daniel V. *Defense System Software Project Managment*. Wright-Patterson AFB OH: Air Force Institute of Technology, 1990.

-----. Class handout, IMGT 626, Software Product Assurance. School of Systems and Logistics, Air Force Institute of Technology, Wright-Patterson AFB OH, Spring Quarter 1993.

Firth, Robert, Vicky Mosley, Richard Pethia, Lauren Roberts, and William Wood. "The Guide to the Classification and Assessment of Software Engineering Tools," *CMU/SEI-87-TR-10*, Pittsburgh PA: Software Engineering Institute (SEI), Carnegie Mellon University, August 1987.

Forte, Gene. "Configuration Management," *IEEE Software*, 9: 79 (May 1992).

Hall, Patrick A.V. "Software Development Standards," *Software Engineering Journal*, 4: 143-147 (May 1989).

Harter, Richard. "Object Oriented Software Configuration Management," *Dr. Dobb's Journal*, 16: 36-71 (October 1991).

McCarthy, Rita. "Applying the Technique of Configuration Management to Software," *Tutorial: Software Configuration Management*. 263-268. New York: IEEE Press, 1980 (EHO 169-3).

Millradt, Bob. "Configuration Management: How Much Do You Need?", *1990 CASE Outlook*, 2: 6-13 (March 1990).

Paulk, M. C., C. Curtis, and M. B. Chrisses, "Capability Maturity Model for Software," *CMU/SEI-91-TR-24*, Pittsburgh PA: Software Engineering Institutc (SEI), Carnegie Mellon University, March 1991.

Richartz, John. "Software Configuration Management Tools," *UNIX Review*, 8: 87-95 (May 1990).

Roetzheim, William H. *Developing Software to Government Standards*. Englewood Cliffs NJ: Prentice-Hall, Incorporated, 1991.

Software Maintenance & Development Systems (SMDS), Incorporated. *ADC/CM Model 209 User's Guide*. Concord MA: Softwarc Maintenance & Development Systems, Incorporated, March 1993.

----. *Aide-De-Camp, Command Reference Guide*. Concord MA: Software Maintenance & Development Systems, Incorporated, September 1992.

----. *Aide-De-Camp, ADC Tutorial for UNIX Systems*. Concord MA: Software Maintenance & Development Systems, Incorporated, 1992.

----. *Aide-De-Camp, User's Guide*. Concord MA: Software Maintenance & Development Systems, Incorporated, September 1992.

----. *X-ADC, Administrator's Guide*. Concord MA: Software Maintenance & Development Systems, Incorporated, April 1993.

----. *X-ADC, User's Guide*. Concord MA: Software Maintenance & Development Systems, Incorporated, April 1993.

SQL Software Limited. *PCMS Overview, Edition 2.1*. Vienna VA: SQL Software Limited, 1992.

Sweetman, Sherri L. "Utilizing Expert Systems to Improve the Configuration Management Process," *Project Management Journal*, 11: 5-12 (March 1990).

Whitgift, David. *Methods and Tools for Software Configuration Management*. West Sussex UK: John Wiley and Sons, Incorporated, 1991.

Vita

Captain Wayne M. Descheneau was born in Biloxi, Mississippi on 23 February 1965 to an Air Force family. Because of extensive travel, he attended numerous schools and in 1983 he graduated from West High School in Manchester, New Hampshire. That same year he attended Boston University on a four year USAF Reserve Office Training Corps (ROTC) Scholarship. In 1987, he graduated as a Distinguished Graduate in his ROTC program and received a Bachelor of Science degree in Aerospace Engineering. His first assignment in the USAF was to the Directorate of Materiel Management, Ogden Air Logistics Center (ALC) at Hill AFB, Utah. He started as a project engineer for the F-16 Flight Simulator Program where he performed technical evaluation and management efforts related to modifications made to the system. He was later moved to the same position on the E-3 AWACS Air Crew Training Device Program which supported all such US, NATO, and Saudi Arabian systems. He was then chosen to be the lead engineer for the F-15 Flight Simulator Program. His responsibilities included managing the program's engineering staff, developing contract specifications and work requirements, and evaluating and managing multi-million dollar software modification efforts. He played a key role in the development of the F-15C Training System Support Center (TSSC) and statement of work which resulted in reducing modification costs and improving system supportability. In May 1992, Captain Descheneau entered the Software Systems Management program in the School of Systems and Logistics, Air Force Institute of Technology.

Permanent Address: 20 Mulberry Lane
Bedford, New Hampshire 03110

Vita

Captain Neil W. Robinson was born in Rutland, Vermont on 29 December 1965. Although his family was not military, he attended schools in Vermont, New York, and New Jersey during his formative years and in 1984, graduated from Ridgewood High School in Ridgewood, New Jersey. That same year, he entered the United States Air Force Academy as a member of the Class of 1988. Four years later, he graduated with a Bachelor of Science degree in Engineering Mechanics (concentration in Structures). His first assignment in the USAF was to the Directorate of Materiel Management, Ogden Air Logistics Center (ALC) at Hill AFB, Utah. He started as an F-16 Landing Gear system engineer where he drafted engineering specification and qualification requirements, analyzed test data for proposed new and/or modified landing gear systems, and resolved technical problems for depot and field maintenance units. Later he became the lead engineer for the F-111 Flight Simulator Program. As the lead engineer, he was responsible for the technical evaluation and management of US, NATO, and Foreign Military Sales F-111 training device modifications and programs support. This role encompassed developing modification purchase descriptions, statements of work, and development, test, and evaluation criteria, evaluating contractor proposals for technical accuracy, validating engineering drawings, data, and software specifications, and directing design reviews and test efforts. In May 1992, Captain Robinson entered the Software Systems Management program in the School of Systems and Logistics, Air Force Institute of Technology.

Permanent Address: 5248 Cobble Creek Drive
Salt Lake City, Utah 84117

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE THE DEVELOPMENT AND USE OF AN EVALUATION MECHANISM FOR THE ASSESSMENT OF SOFTWARE CONFIGURATION MANAGEMENT TOOLS	5. FUNDING NUMBERS
6. AUTHOR(S) Wayne M. Descheneau, Captain, USAF Neil W. Robinson, Captain, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GSS/LAS/93D-3
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) OO-ALC/TISAC, Hill AFB UT 84056	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This study investigated the development of a mechanism for use in the evaluation of Software Configuration Management (SCM) tools. An examination of applicable DoD standards identified the SCM requirements that could be levied on a development contractor, and a literature search revealed the functionality common to various automated tools. These two sets of information were organized into a matrix, and for each requirement that was met, the intersection on the matrix of the requirement and each functionality used to meet that requirement was checked. In addition to the matrix, the mechanism consisted of general information about a given tool and an area to substantiate each requirement identified as being met by the tool. The evaluation mechanism was then used to assess two commercially available SCM tools: Aide-De-Camp and the Product Configuration Management System. The evaluation mechanism prescribes a method for evaluating complex SCM tools and forces the evaluator to gain intimate knowledge of a tool to effectively assess the tool's merits for a given effort.

14. SUBJECT TERMS configuration management, software engineering, taxonomy, life cycles, computer applications	15. NUMBER OF PAGES 122
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

AFIT Control Number AFIT/GSS/LAS/93D-3

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to determine the potential for current and future applications of AFIT thesis research. Please return completed questionnaires to: DEPARTMENT OF THE AIR FORCE, AIR FORCE INSTITUTE OF TECHNOLOGY/LAC, 2950 P STREET, WRIGHT PATTERSON AFB OH 45433-7765

1. Did this research contribute to a current research project?

a. Yes   b. No

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not researched it?

a. Yes   b. No

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency received by virtue of AFIT performing the research. Please estimate what this research would have cost in terms of manpower and/or dollars if it had been accomplished under contract or if it had been done in-house.

Man Years ______________ $ ______________

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3, above) what is your estimate of its significance?

a. Highly Significant   b. Significant   c. Slightly Significant   d. Of No Significance

5. Comments

______________________   ______________________
Name and Grade   Organization

______________________   ______________________
Position or Title   Address

DEPARTMENT OF THE AIR FORCE
AFIT/LAC Bldg 641
2950 P St
45433-7765

OFFICIAL BUSINESS

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 1006 DAYTON OH

POSTAGE WILL BE PAID BY U.S. ADDRESSEE

Wright-Patterson Air Force Base

AFIT/LAC Bldg 641
2950 P St
Wright-Patterson AFB OH 45433-9905